Diploma Thesis

Engineering Fast Parallel Matching Algorithms

Marcel Birn

June 20, 2012

Supervisors: Dipl.-Math., Dipl.-Inform. Christian Schulz
Prof. Dr. rer. nat. Peter Sanders

Acknowledgement

I would like to thank my supervisors Cristian Schulz and Peter Sanders for their support and the constructive discussion about problems that occurred during this work.

Furthermore I would like to thank Cory Niu and Michael Morante for proofreading this thesis. In particular I would like thank my parents, whose support made it possible for me to study computer science.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, June 20, 2012

Marcel Birn

Abstract

The computation of matchings has applications in the solving process of a large variety of problems, e.g. as part of graph partitioners. We present and analyze three sequential and two parallel approximation algorithms for the cardinality and weighted matching problem. One of the sequential algorithms is based on an algorithm by Karp and Sipser to compute maximal matchings [21]. Another one is based on the idea of locally heaviest edges by Preis [30]. The third sequential algorithm is a new algorithm based on the computation of maximum weighted matchings of trees spanning the input graph. We show for two of these algorithms that the runtime for slight variations of them is expected to be linear. However the experimental results suggest that this is also the case for the unmodified versions. The comparison with other approximate matching algorithms show that the computed matchings have a similar quality or even the same quality. On the other hand two of our the algorithms are much faster.

For two of the sequential algorithms we show how to turn them into parallel matching algorithms. We show that for a simple non optimal partitioning of the input graphs speedups can be observed using up to 1024 processors. For certain kinds of input graphs we see a good scaling behaviour.

Contents

1. Introduction

In this work we discuss the sequential and parallel computation of matchings of graphs. A matching M of a graph is a subset of the edges of the graph such that no two distinct edges of M share a common end vertex.

Matchings are of interest in a huge variety of problems. In some of the earliest applications they can be used to directly describe the solution for a problem. For example matchings can be used to solve a classification problem as described by Thorndike [34] or by Lawler [22]. The problem is to assign N workers to N jobs. Each of those jobs belongs to one of k job categories. Each of theses workers are qualified to some extent for each of the different jobs. Lets measure this qualification by $r(w, j)$. That is the qualification of worker w for the job category j. The assignment of the workers to the jobs should maximize the accumulated qualification. This situation can be represented by a weighted bipartite graph. The workers represent one set of vertices while the jobs represent the other set of vertices. Each worker vertex is connected with job vertices by edges, and the weights of those edges correspond to the qualification of the worker for the corresponding job. A maximum weighted matching of this graph defines an optimal solution for the given problem. The maximum weighted matching assigns each worker to a particular job and ensures that the accumulated qualification is maximized.

Another early application was given by Fujii et al. [16, 22]. They used matchings to compute optimal schedules for processors. In this example we have two identical processors and n jobs. Each of these jobs requires one time step to be computed and there exists a precedence relation between them. Job j_i must be executed before job j_j if $j_i > j_j$, and both jobs can be executed simultaneously if $j_i \sim j_j$. For an optimal schedule we are looking for the maximal number of job pairs that can be executed at the same time. An upper bound for the maximal number of pairs corresponds to the maximum cardinality matching of the graph where the vertices represent the jobs and two vertices are connected by an edge if their corresponding jobs are independent from each other, i.e. they can be executed simultaneously. This graph does not have to be bipartite. In [16] it is also shown that jobs of the resulting job pairs can be interchanged in such a way that there is always one job pair that can be executed before the others, with respect to the precedence relation.

Today the computation of matchings is quite often used to solve a subproblem of a bigger problem. For example when computing solutions for a system of linear equations $Ax = b$. In this case it is often required to select good candidates, so called pivot elements, for the diagonal entries of the matrix. Iterative methods like Gauss-Seidel converge faster if the diagonal entry is larger compared to other entries of its row or column [11]. [11] show that it can be beneficial to choose those pivots by computing a weighted matching of the bipartite graph representing the linear equation.The bipartite

graph is constructed the following way: The rows represent one half of the vertex-set and the other half is represented by the columns. A row-vertex and a column-vertex are connected by an edge if the matrix entry of this row and column is nonzero. The edge weight corresponds to the matrix entry.

Another recent example for the necessity for the computations of matchings is multi level graph partitioning as used by [20]. Multi level graph partitioning is done by *contracting* (coarsening) a graph recursively for several rounds until the graph size falls below a certain threshold. This small graph is then used to compute an initial partition. Starting with this initial partition new partitions are computed for each contracted graph in reverse order by uncontracting the graphs [20]. The contraction is done by computing a weighted matching M of the graph during each round and afterwards contracting the edges of M. Edge ratings are used as the weights of the edges. The ratings specify for each edge how suitable it is for a contraction. Contracting M means that each edge $e \in M$ is contracted to a single vertex. In [20] newly formed multi-edges are treated as a single edge but their edge weights are summed up. Furthermore the weight of each new vertex is the sum of the weights of the two end vertices of the contracted edge. For contracted graphs we also have to store the information that is necessary to uncontract the graph.

Interestingly the maximum cardinality matching and maximum weighted matching problems can be solved in polynomial time, where on the other hand the similar sounding problem of finding maximum independent sets is NP-complete for planer graphs [6]. An independent set I is a subset of the vertices of a graph, such that no two vertices of I are adjacent. The first to show that maximum weighted matchings can be computed in polynomial time was Edmonds [13]. For an $O(n^3)$ implementation see [22]. Gabow improved this runtime later to $O(n(m + n \log n))$ [17].

However there are problems where we have huge graphs as inputs (e.g. graph partitioning) or where it is not even necessary to achieve optimal results. In such cases we might be satisfied with approximation algorithms for the maximum cardinality and maximum weighted matchings problems. Most importantly these algorithms achieve far better runtimes and are much simpler to implement. Karp and Sipser present in [21] a linear-time approximation algorithm for the maximum cardinality matching problem.

In case of the maximum weighted matching problem the simplest approximation algorithm is probably a greedy algorithm. This algorithm continuously adds the heaviest remaining edge to the matching and achieves a runtime of $O(m \log n)$ [4]. Another greedy algorithm, by Preis, which adds locally heaviest edges to the matching achieves a runtime of $O(m)$ [30]. Both greedy algorithms have an approximation factor of $1/2$. Other $1/2$-approximation algorithms are PGA and GPA [10, 26]. They have runtimes of $O(m)$ and $O(m \log n)$ respectively. There are also $2/3 - \varepsilon$ approximation algorithms, e.g. in [29] with a runtime of $O(m \log(1/\varepsilon))$.

Recently there has also been some work on parallel matching algorithms [28, 19, 25, 7], most of them are based on the greedy algorithm by Preis.

1.1. Our Contribution

We present three sequential algorithms: Two approximation algorithms for the maximum weighted matching problem. One of them is based on Preis' idea of adding locally heaviest edges to the matching. Another one computes maximum weighted matching of sets of trees spanning the input graph. The third sequential algorithm is a variation of the algorithm of Karp and Sipser to compute approximate maximum cardinality matchings. We provide theoretical results for their expected runtimes and the quality of the worst case solutions. Additionally, we performed experiments on a wide variety of graphs to see how those algorithms perform in practice with regard to their runtime and the quality of their solutions compared to each other and other algorithms.

We also present parallel versions of the two sequential algorithms for the weighted matching problem and evaluate how well their implementations scale in practice.

1.2. Outline

Chapter 2 introduces the terminology that we are using in this thesis, Chapter 3 discusses related work. The sequential algorithms are described in Chapter 4, as well as their theoretical and experimental results. Chapter 5 talks about the two parallel algorithms and their experimental results. Finally Chapter 6 concludes this work by summarizing our main results and talks about possible future work.

2. Terminology

2.1. Graphs

An *undirected graph* $G = (V, E)$ consists of a set V of *vertices* and a set E of *edges*. An edge $e = \{a, b\}$ connects two vertices a and b of V. We use the variable n to specify the size of V, that is the number of vertices of the graph and $m = |E|$ is used to specify the number of edges of the graph G. When talking about a graph G we implicitly say that its vertex set is V and its edge set is E.

In general we allow *multi-graphs*. These are graphs which allow multiple edges between two vertices and *self-loops* (both end vertices of an edge are the same).

And edge $e \in E$ is *incident* to a vertex $a \in V$ if a is an end vertex of e. Two vertices a and b are *adjacent* if both are connected by an edge. In a similar way we say that edges e and e' are *adjacent* if they have a common end vertex. The number if incident edges of a vertex v is called the *degree* d of v. A vertex is called a degree d vertex if its degree is d.

A *weighted graph* $G = (V, E, c, \omega)$ is a graph as specified before but with two additional functions. The function $c : V \to \mathbb{R}^+$ assigns costs to the vertices and $\omega : E \to \mathbb{R}^+$ assigns a weight to each edge.

A *complete graph* K_n is a graph with n vertices and one edge between each pair of vertices. A *bipartite graph* is a graph where the set V of vertices can be divided into two sets of vertices V_1 and V_2 such that $V = V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$ and there are no edges connecting two vertices of V_1 or two vertices of V_2.

2.2. Matchings

A *matching* of a graph G is a subset $M \subseteq E$ of the graph's edges such that no two edges of M share a common end vertex. We say that the edges of M are *matched* and a vertex is *matched* if it is an end vertex of a matched edge otherwise it is *unmatched*. An edge is *unmatched* if it is not an element of M.

A matching M is *maximal* if there is no edge $e \in E \setminus M$ such that $M \cup \{e\}$ is still a valid matching.

A *maximum cardinality matching* M of a graph G is a matching with $|M| \geq |M'|$ for any other matching M' of G. We say that M is a *perfect* matching if all vertices of the graph are matched ($2|M| = n$). Obviously perfect matchings only exists for graphs with an even number of vertices.

The cardinality of a maximal matching is at least half the size of a maximum cardinality matching.

The *weight* $w : 2^E \to \mathbb{R}^+$ of a matching M is the sum of the weights of each edge of M:

$$w(M) = \sum_{e \in M} \omega(e)$$

A *maximal weighted matching* is a weighted matching where no more edges can be added. A *maximum weighted matching* M is a matching with $w(M) \geq w(M')$ for any other matching M'.

We also consider approximations of maximum cardinality matchings and maximum weighted matchings. In those cases we try to achieve the optimal results, but do not require it. However the results must be maximal.

2.3. Parallel Algorithms

Throughout this thesis we only consider parallel algorithms for *distributed memory systems*. That is we have a number of p processors and each of these processors has its own local memory. No processor is directly able to access the local memory of another processor, instead processors have to send messages to each other to exchange information. The processors can communicate with each other using a network that connects them [12, p. 574].

Bulk Synchronous Parallelism (BSP) can be considered as computation model for parallel algorithms but also as an approach how to develop parallel algorithms [12, p. 192]. When we talk about BSP in this work we only consider it as an approach how to write parallel algorithms. A BSP-algorithm divides its execution into several super steps. At the start of each super step the processors perform local computations independent from each other. As soon as they are finished with their computations they start a communication phase where they send messages to other processors and also receive messages. At the end of each super step there is a synchronization barrier to ensure that no processors continues to perform local computations before the other processors have finished their computations and communications [12, p. 192].

In the case of parallel graph algorithms the input graph $G = (V, E)$ is usually partitioned into several subgraphs $G_p = (V_p, E_p)$, with one subgraph for each processor p. The subgraphs G_p combined represent the whole input graph G. In our case the subgraphs are defined by assigning to each process a subset V'_p of vertices of V, such that those subsets are disjoint but the union of them is equal to V. The subgraph G_p is then defined by the set E_p of edges incident to a vertex of V'_p and the vertex set $V_p = V'_p \cup \{u \in V | u \text{ end vertex of an edge } \in E_p\}$. Obviously it is possible that for an edge $e = \{u, v\} \in E_p$ only one vertex is an element of V'_p and the other vertex is not. We call such an edge a *cross edge* and vertices $\in V'_p$ are called *local vertices* and vertices $\in V_p \setminus V'_p$ are called *ghost vertices*. Edges that connect two local vertices are called *local edges*.

Each cross edge $e = \{u, v\}$ connects a processor with another processor, the two end vertices are assigned to different processors. We write $p(v)$ to identify the processor a vertex v is assigned to.

Usually the main reason why someone is interested in parallel algorithms is to compute the solution of a problem a lot faster. But in the case of distributed programs it also allows us to use a lot more resources other than processing power, e.g. main memory.

To see how well a parallel algorithm performs/scales (i.e. how much faster it is), we look at the *speed up* $S = {}^{T_s}/_{T_p}$. That is the runtime T_s of the sequential algorithm divided by the parallel runtime T_p. In case of p processors we would hope that the parallel versions is p times faster than the sequential version. In [3] Amdahl made the observation that the fraction α of a program that cannot be computed in parallel is constant and showed how well a program performs for different sizes of α. This lets us directly compute the parallel runtime T_p of a algorithm on a system with p processors:

$$T_p = \alpha \; T_s + (1 - \alpha) \; \frac{T_s}{p}$$

This results in the following speedup for a parallel program:

$$S = \frac{T_s}{T_p} = \frac{T_s}{\alpha \; T_s + (1 - \alpha) \; \frac{T_s}{p}} = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

This formula is also known as *Amdahl's Law* [12, p. 53]. One can directly derive from it that the speedup cannot exceed $1/\alpha$.

But Gustafson made in [18] the observation that in practice people are not interested in getting their computations done faster, but instead use a larger number of processors to solve the problem for a bigger input size in the same time. Therefore he suggested to fix the runtime and not the problem size. Another observation was that the time spent on the sequential part of the program does not increase with the problem size, instead it remains constant. This lets us split the runtime T_p of the parallel program (using p processors) into a sequential part t_s and a parallel part t_p: $T_p = t_s + t_p$. Using this information we get for the sequential runtime of the program $T_s = t_s + p \; t_p$. Scaling $t_s + t_p$ to 1 gives us a speedup of

$$S = \frac{T_s}{T_p} = \frac{t_s + p \; t_p}{t_s + t_p} = t_s + p \; t_p = t_s + p \; (1 - t_s) = p - t_s(p - 1)$$

In this case t_s represents the sequential fraction of the parallel runtime T_p. This formula is also known as *Gustafson's Law* [12, p. 819].

Those two laws suggest two different approaches to measure the performance of a parallel algorithm. The first is known as *strong scaling* and based on Amdahl's law [12, p. 1127]. In this case we fix the problem size and increase the number of processors. In the optimal case the speed up would increase linear with the number of processors (doubling the number of processors should halve the runtime).

The other approach is known as *weak scaling* and is based on Gustafson's law [12, p. 1127]. This time we increase the problem size by the same fraction as we increase the number of processors, hoping that the problem size for each processor remains constant. In this case we should see a constant runtime of the algorithm, in case of optimal scaling.

3. Related Work

3.1. Greedy Weighted Matchings

One of the earliest and simplest approximation algorithms for the weighted matching problem is the *greedy algorithm* shown in Algorithm 3.1.1. It simply adds the heaviest remaining edge to the matching and removes this edge and adjacent edges until no edges are left. An interesting property of this algorithm is that it guarantees an 1/2-

Algorithm 3.1.1 Compute an approximate weighted matching

greedy$(G = (V, E))$:

1: $M_{\text{greedy}} = \emptyset$
2: **while** $E \neq \emptyset$ **do**
3: $\quad e = \{a, b\}$ edge with highest weight in E
4: $\quad M_{\text{greedy}} = M_{\text{greedy}} \cup \{e\}$
5: $\quad$ remove all edges incident to a or b from E

approximation of the optimal result. For each edge e that is added to the matching we miss at most two edges (both adjacent to e) of the optimal result. However the combined weight of the missed edges is at most twice the weight of the edge e [4].

The obvious best runtime for the greedy algorithm is $O(m \log n)$, for non multi-graphs, when using comparison based sorting to sort the edges by decreasing weight.

Preis showed that a variation of the greedy algorithm provides the same 1/2-approximation factor as the greedy algorithm but with a linear runtime $O(m)$ [30]. Preis' algorithm

Algorithm 3.1.2 Compute an approximate weighted matching

LAM$(G = (V, E))$:

1: $M_{\text{LAM}} = \emptyset$
2: **while** $E \neq \emptyset$ **do**
3: $\quad e = \{a, b\}$ a locally heaviest edge of E
4: $\quad M_{\text{LAM}} = M_{\text{LAM}} \cup \{e\}$
5: $\quad$ remove all edges incident to a or b from E

LAM is shown in Algorithm 3.1.2. The difference to the greedy algorithm is that instead of choosing the heaviest remaining edge *LAM* chooses a locally heaviest edge. An edge is a *locally heaviest* edge, if it is heavier than any of its remaining adjacent edges. The idea to choose locally heaviest edges will be one of the main ideas of the following chapters.

The LAM-algorithm is a more general version of the greedy algorithm. Obviously LAM simulates the greedy algorithm if it chooses the heaviest remaining edge each round. The

heaviest remaining edge must be a locally heaviest edge. In case of unique edge weights the LAM-algorithm computes for each possible run (the order in which edges are added to the matching) the same matching (Theorem B.1), therefore it computes the same matching as the greedy algorithm and the weight of the matching is at least half the weight of an optimal matching.

Preis also showed this 1/2-approximation for arbitrary edge weights.

3.2. Karp-Sipser

Karp and Sipser propose in [21] a simple algorithm for the maximum cardinality matching problem. Despite its simplicity Karp and Sipser show that the algorithm is expected to give near optimal results for random graphs. The basic structure of this matching algorithm is shown in Algorithm 3.2.1. We assume that the provided graph initially does not contain any degree zero vertices. The algorithm is organized in rounds and in each of these rounds the algorithm looks for degree one vertices. If there is such a vertex its incident edge is added to the matching otherwise an edge is chosen at random and added to the matching. Using a list to keep track of degree one vertices, it is easy to implement

Algorithm 3.2.1 Compute a matching using the Karp Sipser heuristic [21]

ks_matching$(G = (V, E))$:

```
1:  M = ∅
2:  while G not empty do
3:    if G contains degree one vertex then
4:      v one of the degree one vertices
5:      e = {v, u} the edge incident to v
6:      M = M ∪ {e}
7:      remove e and all adjacent edges from G, also remove all degree zero vertices
8:    else
9:      pick an edge e = {v, u} at random
10:     M = M ∪ {e}
11:     remove e and all adjacent edges from G, also remove all degree zero vertices
12: return M
```

ks_matching in *linear time*. Whenever an edge $e = \{u, v\}$ is added to the matching the edge e and its adjacent edges are removed from the graph. For each removed adjacent edge we have to check the degree of the remaining end vertex. If the degree is one we add the vertex to the list of degree one vertices. The runtime for removing a *matched edge* is $O(\deg(u) + \deg(v))$ this gives a total runtime of $O(m)$.

Selecting a degree one vertex is called a *degree 1 reduction* by Magun in [24]. As proven by Karp and Sipser a degree 1 reduction does not change the optimality of a maximum cardinality matching [21]. In the same paper Karp and Sipser also showed that for a degree two vertex u, with incident edges e and e', either e or e' is an edge of an maximum cardinality matching M. This observation is called a *degree 2 reduction*

by Magun. Karp and Sipser also propose another algorithm which additionally uses a degree 2 reduction if there is no degree one vertex.

Other variants of those two algorithms are studied in [24] and [8]. Those algorithms are still based on degree 1 and degree 2 reductions but instead of choosing a random edge, if there is no degree one or degree two vertex, another heuristic is used. For example choosing a remaining vertex of minimal degree and an incident edge of this vertex.

3.3. Optimal Weighted Matchings of Trees

Solving the maximum cardinality matching problem for trees can be easily done in linear time using the degree 1 reduction from the Karp and Sipser algorithm. There is always a degree one vertex in a tree and hence we get a optimal solution for the maximum cardinality problem.

Solving the maximum weighted matching problem for a tree can also be done in linear time using a dynamic programming approach [32]. It is easier for this algorithm to treat the trees as directed trees starting at a distinct root vertex r.

The *optimal solution of a tree* T can be computed by using the optimal solutions of the *subtrees* of T. For each subtree s of T there are two possible matching results. Either the incoming edge to s is matched or it is not. The incoming edge of a subtree s is the edge which connects s with its parent. Those two possible results can be represented by the corresponding accumulated weights of the result of the subtree s. From now on we may also talk about the vertex s, in this case we mean the root of the subtree s.

If the incoming edge of s is matched we cannot match any of the outgoing edges of s. Thus the result for this case is the weight of the incoming edge plus the weights of all subtrees of s for the case that the incoming edges of those subtrees are not matched.

If the incoming edge of s is not matched we can match one of the outgoing edges of s. Thus the result for this case is the sum of the weights of all subtrees of s depending whether the incoming edge is matched.

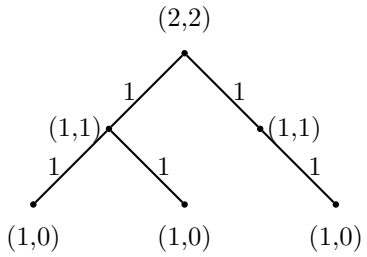


Figure 3.3.1.: Simple example for computing matchings in trees.

An example is shown in Figure 3.3.1. Each edge has a weight of 1. Next to each vertex of the tree is a pair. The first entry specifies the weight at the vertex's subtree (including the incoming edge) for the case that the incoming edge is matched and the second entry gives the weight for the case that the incoming edge is not matched. There

is a special case for the root of the tree. Obviously this vertex does not have any incoming edges, thus we say that the incoming edge has weight 0. In Figure 3.3.1 the optimal accumulated weight is 2. To get the edges of the matching we walk through the tree from top to bottom and add edges according to the weight-pair at each node and the edges that already have been added.

Algorithm 3.3.1 shows the general structure how to compute an optimal matching for a given tree. At first we fill a table O storing for each vertex/subtree which outgoing edge will be matched if the incoming edge is not matched. The outgoing edges stored in O are represented by the other end vertex of the edge, i.e. an entry $O[v]$ represents the edge $\{v, O[v]\}$. Initially each entry of O references a non existent vertex.

Algorithm 3.3.1 Compute maximum weighted matching of a tree

weighted_matching_of_tree(T):

1: *initialize*(O, T) // *One entry for each node of* T
2:
3: *fill_subtree_table*(T, O)
4:
5: $M = \emptyset$
6: *add_matching_edges*(*root*(T), false, O, M)
7:
8: **return** M

Algorithm 3.3.2 describes the computation of the subtree table O. At first we initialize another table S. This table stores for each vertex of T a pair containing the two possible accumulated weights of the corresponding subtree. The first entry represents the case that the incoming edge is matched and the second entry the case that the incoming edge is not matched. Afterwards we compute a breadth first traversal (BFS-traversal) of the tree T. This traversal is used to fill both tables O and S *bottom up*, by walking through the BFS-traversal in reverse order. The algorithm distinguishes the cases that a vertex is a leaf or an inner vertex.

In the case of inner vertices we have to be careful when computing outgoing edges. Because if the incoming edge is not matched we can match one of the outgoing edges, but we *do not have to*. Figure 3.3.2 shows an example were in one case (vertex v) no outgoing edge is matched. This inner vertex will not be matched at all. In such a case the corresponding entry of the table O references a non existent vertex, to indicate that no outgoing edge will be matched.

To check if an outgoing edge will be matched, in the case that the incoming edge is not matched, we have to verify that the accumulated weight increases. An outgoing edge increases the accumulated weight by the difference c of the two weights of its subtree: matched incoming edge (w_1) and incoming edge not matched (w_2).

$$c = w_1 - w_2$$

We want that c is at least 0, i.e. it does not matter whether we match the incoming edge or not. Also an edge is only matched if its increment is larger than the best increment

Algorithm 3.3.2 Fill subtree table of tree

fill_subtree_table(T, O):

```
 1: initialize(S, T) // Results of subtrees of T
 2: bfs_traversel = create_bfs_traversal(T)
 3:
 4: // Traverse the tree bottom up
 5: for n ∈ reverse(bfs_traversel) do
 6:    w = incoming_weight(n)
 7:
 8:    if out_degree(n) == 0 then
 9:       S[n] = (w, 0)
10:    else
11:       w_i = w // Weight if the incoming edge is matched
12:       for all o ∈ outgoing_neighbors(n) do
13:          w_i = w_i + S[o].second
14:
15:       w'_i = 0 // Weight if the incoming edge is not matched
16:       b    = 0 // Best weight difference
17:       for all o ∈ outgoing_neighbors(n) do
18:          c = S[o].first - S[o].second
19:          if c ≥ b then
20:             w'_i    = w'_i + (S[o].first - b)
21:             O[n] = o
22:             b    = c
23:          else
24:             w'_i = w'_i + S[o].second
25:
26:       S[n] = (w_i, w'_i) // Set the result for the current subtree
```

b observed so far. This check is done in line 19 of Algorithm 3.3.2. If an outgoing edge is matched we have to compute the value of the new accumulated weight, i.e. we have to subtract b and add the weight w_1. Subtracting b from the accumulated weight corresponds to the case that we do not match the corresponding edge of b.

Algorithm 3.3.3 describes the computation of matched edges. The first important observation for this function is, that the second entry of the root-vertex entry of the table S (computed by Algorithm 3.3.2) always specifies the *maximum* of the two values. The first entry of this pair is just the sum of all weights of subtrees such that the incoming edge of them is not matched (the incoming edge of the root vertex has weight 0). But the second entry is either equal to this value or larger in the case that the accumulated weight is improved by matching one of the outgoing edges of the root. Thus we can say when calling *add_matching_edges* for the root-vertex, that the incoming edge is not matched and we do not need any further information from the table S, we just have

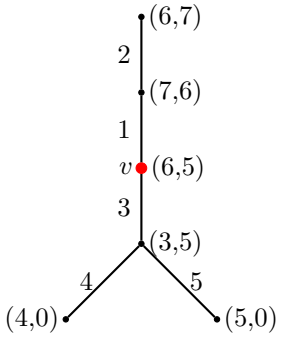


Figure 3.3.2.: Unmatched internal vertex.

Algorithm 3.3.3 Add matched edges to the matching

add_matching_edges(root, added_incoming, O, M):

```
1: if added_incoming then
2:   for all n ∈ outgoing_neighbors(root) do
3:     add_matching_edges(n, false, O, M)
4: else
5:   for all n ∈ outgoing_neighbors(root) do
6:     if n == O[root] then
7:       M = M ∪ {edge(root, n)}
8:       add_matching_edges(n, true, O, M)
9:     else
10:      add_matching_edges(n, false, O, M)
```

to know which outgoing edge is matched if the incoming edge is not. Apart from this Algorithm 3.3.3 is just a *depth first walk* through the tree.

The total runtime of Algorithm 3.3.1 is in $\Theta(n)$, where n is the number of vertices of tree T. The runtime of Algorithm 3.3.1 is the sum of the runtimes of Algorithm 3.3.2 and Algorithm 3.3.3. Obviously a lower bound for the runtime is $\Omega(n)$, each vertex is visited at least once.

The main loop of Algorithm 3.3.2 visits each vertex once and for each vertex we do some computations for the outgoing edges. But the time spent for each outgoing edge has a constant upper bound of o. All the other computations done for the vertices only require a constant amount of time. This gives an upper bound of $c + o$ out_deg(v) for the time spent on each vertex v. Therefore the total runtime of the main loop of Algorithm 3.3.2 is at most:

$$\begin{aligned}\sum_{v \in V} c + o \text{ out\_deg}(v) &= \left(\sum_{v \in V} c\right) + \left(o \sum_{v \in V} \text{out\_deg}(v)\right) \\ &= cn + o\ (n-1) = O(n)\end{aligned}$$

A tree has exactly $n-1$ edges. The computation of the BFS-traversal and the initialization of the table S can both be done in $O(n)$. Thus the total runtime of Algorithm 3.3.2 is in $O(n)$.

With a similar argumentation (each vertex is visited once and we look at each outgoing edge once) we see that Algorithm 3.3.3 has a runtime of $O(n)$.

3.4. Global Paths Algorithm

The global paths algorithm (GPA) is a matching algorithm presented by Maue and Sanders in [26]. This is the main algorithm we use for comparison with the algorithms presented in this thesis.

Algorithm 3.4.1 Compute a weighted matching

GPA$(G = (V, E))$:

1: $M = \emptyset$
2: $E' = \emptyset$ // *Collection of paths and cycles*
3: **for all** edges $e \in E$ in decreasing order **do**
4: add edge e to E' if e is applicable
5: **for all** paths and cycles P in E' **do**
6: $M = M \cup max\_weighted\_matching(P)$
7: **return** M

During the first phase of GPA, shown in line 3 of Algorithm 3.4.1, edges are added to a set E' in decreasing order if they are *applicable*. That is similar to the greedy algorithm. An edge is *applicable*, if the edge connects two different paths, of E', at their end vertices or if it connects the two end vertices of an odd length path – a cycle of even length is created. Initially vertices are considered as paths of length zero. The resulting set E' consists of paths and cycles. This is similar to the PGA and PGA' algorithms from [9, 10] they both grow paths at first and then compute weighted matchings of those paths. In the second phase of the GPA algorithm a maximum weighted matching is computed for each path and cycle of E'. This can be done in linear time using dynamic programming [26].

The resulting matching produced by a single round of GPA is not necessarily maximal as is shown in Figure 3.4.3. A single round would miss the edge h. Maue and Sanders propose to run the GPA algorithm for three rounds, experimental results have shown that this usually produces maximal results [26]. Drake and Hougardy propose in [10] to just add any remaining edges until the result is maximal, i.e. no edges are remaining.

Like the greedy algorithm the global paths algorithm has a guaranteed *approximation factor* of $1/2$, but performs a lot better in practice [26]. The runtime of GPA is $O(m \log n)$ for comparison based sorting. The first phase has runtime $O(m \log n)$ and the second phase can be computed in linear time [26].

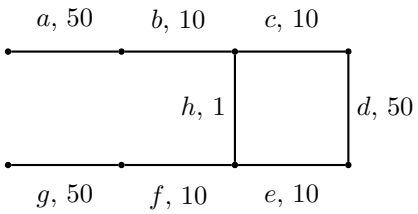


Figure 3.4.3.: A single round of GPA does not return a maximal matching.

3.5. Parallel Matching Algorithms

There has been some work on parallelizing matching algorithms, although quite a lot of the sequential matching algorithms are inherently sequential. For example in the greedy algorithm from Algorithm 3.1.1 the order in which the edges are visited is predefined and whether an edge is matched depends on the previously matched edges.

Patwary et al. describe in [28] a method how to parallelize the algorithm from Karp and Sipser to find approximate maximum cardinality matchings. Instead of distributing the vertices of a graph over the available processors they decided to distribute the edges. We are not going into more detail of this algorithm since we are more interested in weighted matchings than in cardinality matchings.

Hoepman suggests in [19] a parallel matching algorithm based on Preis' idea to add locally heaviest edges to the matching. Hoepman assumes that there is one processor for each vertex $v \in V$ of the graph G, of course in practice that is not a valid assumption. Each vertex v is assigned to one processor and knows all its incident edges, to be able to compute its heaviest incident edge $e = \{v, u\}$. The other end vertex u of e is called the *candidate* of vertex v. If the vertex v is the candidate of the vertex u, then we know that the edge e is a locally heaviest edge. Therefore to decide on locally heaviest edges, vertices just have to send messages to their candidates. Those messages are called *request messages*.

Algorithm 3.5.1 describes this approach, but the notation is the same as the one chosen by [25]. The set R is used to store the vertices from which request messages have been received. The set S specifies all the adjacent vertices that have not been matched. We also call adjacent vertices *neighbors*. Initially S is set to the *neighborhood* $N(v)$ of the vertex v, that is the set of all adjacent vertices of v in the graph G. Each neighbor of v that has been matched is removed from the set S. The function $H_S(v)$ returns the adjacent vertex $u \in S$ of v such that the edge $\{v, u\}$ is the heaviest edge of all edges from vertex v to a vertex in S. Therefore it is the *candidate* of v, of all the remaining vertices. If there is no adjacent edge of v in S then $H_S(v)$ returns *null*.

At first each processor computes a matching candidate $c = H_S(v)$ and sends the message $\langle req \rangle$ to c, if $c \neq null$. Sending or receiving a message from a vertex is short for saying to send/receive a message to/from the processor responsible for this vertex. Sending a message $\langle req \rangle$ from vertex v to a vertex u tells u that the vertex v wants to match with u. If $v = H_S(u)$ then u knows that the edge to v is a locally heaviest edge.

Algorithm 3.5.1 Compute a matching in parallel – one processor for each vertex v [25]

hoepman_matching$(v \in V)$:

```
 1: R = ∅
 2: S = N(v)
 3: c = H_S(v)
 4: if c ≠ null then
 5:    send message ⟨req⟩ to c
 6: while S ≠ ∅ do
 7:    receive message m from some vertex u
 8:    if m = ⟨drop⟩ then
 9:       S = S \ {u}
10:       if u = c then
11:          c = H_S(v)
12:          if c ≠ null then
13:             send message ⟨req⟩ to c
14:    else
15:       R = R ∪ {u}
16:    if c ∈ R then
17:       for all w ∈ S \ {c} do
18:          send message ⟨drop⟩ to w
19:       S = ∅
20: return c
```

Afterwards each processor starts to communicate with the other processes, until all vertices are removed from the set S. It is possible that S gets empty but v is not matched, in this case v is an unmatched vertex.

Within the while loop each vertex v at first receives a message from an arbitrary source, actually those messages must come from neighboring vertices. Afterwards v checks the kind of message it just received. If it is a $\langle drop \rangle$-message it removes the sending vertex from its set S, a $\langle drop \rangle$-message indicates that the neighbor is matched. After removing the neighbor from S a new candidate $c = H_S(v)$ is computed if S is not empty and a $\langle req \rangle$-message is send to the new candidate.

If the received message is of type $\langle req \rangle$ then the sending vertex u is added to the set R. In the last step during one round of the while-loop we check if $c \in R$. If c is in R we know that the edge to c is a locally heaviest edge. Hence we send $\langle drop \rangle$-messages to all of v's neighbors but c, and set S to $\emptyset$. The matched partner is not necessarily the vertex from which we just received a message. If the message received is a $\langle drop \rangle$-message a new candidate is computed and it is possible that we have already received a $\langle req \rangle$-message from this vertex. If the final value of c is *null* then there is no matched edge incident to v and v is an unmatched vertex otherwise the edge $\{v, c\}$ is part of the matching.

As shown in [19] the weight of the resulting matching is at least half the weight of the optimal solution. One can prove this by showing that the temporal order in which processors decide on locally heaviest edges is a valid order in which Preis' LAM algorithm

adds edges to the matching. The total number of sent messages is at most $2m$, no more then 2 messages are sent per edge. One message from each end vertex [19].

As previously mentioned Hoepman's method is not suitable for real world applications since there are usually a lot more vertices than processors available. But both Manne et al. [25] and Catalyürek et al. [7] have described parallel implementations based on Hoepman's algorithm.

Both algorithms have in common that they use pre-distributed graphs to make sure that about the same number of vertices ($\sim n/p$) is assigned to each processor while minimizing the number of cross edges. The number of cross edges is an indicator for the total number of communication operations. Each processor also stores ghost vertices to handle cross edges.

The algorithm by Manne and Bisseling [25] is based on the following sequential algorithm. The sequential algorithm at first computes for each vertex v the incident edge with the highest weight, represented by the matching candidate $c(v)$. If $c(c(v)) = v$ then edge $\{v, c(v)\}$ is a locally heaviest edge and we add it to the matching M, also the two vertices v and $c(v)$ are added to a queue D. This queue is used to keep track of matched vertices that have not been removed from the graph. Afterwards we loop through the queue D. In each round we remove one vertex v from D and update the information of each neighbor. The edges to the adjacent vertices x are removed and the matching candidate of vertex x is recomputed if $c(x) = v$. If this results in a new found locally heaviest edge ($c(c(x)) = x$) we add the edge $\{x, c(x)\}$ to M and x and $c(x)$ to D. The loop, and hence the algorithm, terminates as soon as D becomes empty.

The parallel implementation at first runs on each processor the sequential algorithm for the vertices assigned to this processor until no more locally heaviest edges can be found. Then a communication phase starts which is similar to Hoepman's algorithm, this results in a BSP-style algorithm. If a boundary vertex v (i.e. a vertex adjacent to a ghost vertex) has been matched in the sequential part, a message is sent to each processor where v is a ghost vertex to inform those processors that v is matched. In case that a vertex v wants to match with a ghost vertex u this request information is sent to the processor $p(u)$. If on processor $p(u)$ u wants to match with v a new locally heaviest edge is found and added to the matching. The vertices v and u are added to the queue D. After the communication phase has finished the sequential phase starts again if D is not empty. This implementation was tested using up to 32 processors and the experiments showed that the algorithm scales well for this number of processors [25].

Manne and Bisseling also point out an interesting connection between computing weighted matchings based on locally heaviest edges and a parallel implementation for the maximal independent set problem by Luby [23]. We get back to the connection to Luby's algorithm later in this work.

The parallel algorithm by Catalyürek et al. [7] uses two interleaved loops. The inner loop iterates over internal vertices and the outer loop iterates over boundary vertices. Similar to Manne's algorithm the inner loop uses a queue to keep track of matched vertices that have to be processed. The outer loop tries to match boundary vertices and creates messages if necessary. Unlike the Hoepman algorithm three kinds of messages are used: Request messages, succeed messages (a vertex has been matched) and failed

messages (a vertex cannot be matched). If there are several messages from one processor p_i to a processor p_j those message are bundled in a single big message, thus reducing the total amount of messages.

Catalyürek et al. also claim that their implementation is asynchronous, although it is not clear how this is done from the information provided in their paper. As they show in their paper the algorithm scales well for thousands of processors (up to 16384) using grid graphs and bipartite graphs as the input.

4. Sequential Algorithms

4.1. Local Max Algorithm

Now we introduce another variation of Preis' algorithm to compute approximate maximum weighted matchings. We call it *local max algorithm*. Algorithm 4.1.1 shows the basic outline of this variant. As usual, the algorithm gets a graph $G = (V, E)$ as the input, and returns a maximal matching of this graph. The weight of the returned matching is at worst $1/2$ the weight of an optimal result. The difference to Preis' LAM algorithm

Algorithm 4.1.1 Compute an approximate weighted matching

local_max$(G = (V, E))$:

1: $M = \emptyset$
2: **while** $E \neq \emptyset$ **do**
3: $\quad L =$ *get_locally_heaviest_edges*(G)
4: $\quad M = M \cup L$
5: $\quad$ *remove_matched_edges*(G, L)
6: **return** M

is, instead of selecting a single locally heaviest edge in each round, we select each edge that is a locally heaviest at the start of the round. Those edges are then added to the matching and afterwards removed from the graph as well as all adjacent edges. There is one problem with this approach of adding edges, it only works if we assume that no two adjacent edges have the same weight. Otherwise it could happen that two adjacent edges are added to the matching. To solve this problem we use a mechanism to *break ties* in the case that there are two adjacent edges with the same weight. Thus, the algorithm implies a graph with unique edge weights. Techniques to break ties might be based on the IDs of the edges or, if there are no edge IDs, one might use vertex IDs, if those exist. In the case of edge IDs the edge with the higher ID wins. In the case of vertex IDs we have two edges $e = \{u, v\}$ and $f = \{w, x\}$ with the same weight. To break the tie we can use $\max\{u, v\} > \max\{w, x\}$ or $\max\{u, v\} = \max\{w, x\}$ and $\min\{u, v\} > \min\{w, x\}$ [25]. Those are just two options, it is not hard to think of other possibilities. From now on we assume that we have a graph with unique edge weights.

There are several advantages to this approach. It is really easy to implement without any complex data structures, if one is satisfied with a runtime of $O(m')$ for a single round. Here, m' is the number of remaining edges at the start of each round. Such an implementation is described in Section 4.1.1. Although in the *worst case* this yields an algorithm with runtime $O(m^2)$. A worst case example is shown Figure 4.1.1. The example shows a path of length m with increasing edge weights. During each round

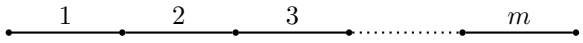


Figure 4.1.1.: $O(m^2)$ runtime for local max algorithm.

Algorithm 4.1.1 would only select a single edge (the heaviest one) and thus require a total of $m/2$ rounds, the matched edge and the one adjacent edge will be removed. However the experimental results in Section 4.4 suggest that far more than half of the remaining edges are removed during each round. Hence in practice the expected runtime is in $O(m)$. A slight variation of the algorithm also provides this expected value in theory. Although this variation is not suitable for maximum weighted matchings. We get to this in more detail in Section 4.1.2.

Another nice fact about Algorithm 4.1.1 is that it provides us with a simple BSP-style parallelization approach. We will talk about the parallelization in Section 5.1.

Now lets have a look at a more interesting example than the one from Figure 4.1.1. In the top picture of Figure 4.1.2 we see a graph with 11 vertices and 16 edges with weights assigned to them. During the first round of Algorithm 4.1.1 the edges $\{1, 2\}$ and $\{4, 6\}$

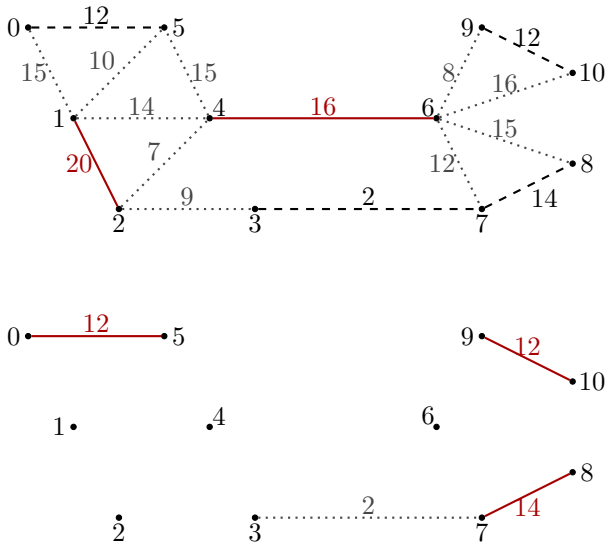


Figure 4.1.2.: Example for Algorithm 4.1.1. Top picture first round, bottom picture second round.

are selected (lets assume that the tie breaking chooses $\{4, 6\}$ instead of $\{6, 10\}$). At the end of the first round those edges and all adjacent edges will be removed. The resulting graph is shown in the bottom picture of Figure 4.1.2. During the second round the edges $\{0, 5\}$, $\{7, 8\}$ and $\{9, 10\}$ are selected. Removing them and the adjacent edges results in an empty graph. The algorithm matched the 5 edges $\{1, 2\}$, $\{4, 6\}$, $\{0, 5\}$, $\{7, 8\}$ and $\{9, 10\}$ with a total weight of 74 and one unmatched vertex is left.

4.1.1. Implementation Details

Our implementation of Algorithm 4.1.1 is shown in Algorithm 4.1.2. As we have already mentioned, we do not need any complex data structures. It is sufficient to represent the graph by an edge-array. Further we need two more arrays of size n to keep track of matched vertices and the heaviest incident edge of a vertex. An edge is represented

Algorithm 4.1.2 Compute an approximate weighted matching

local_max_implementation$(G = (V, E))$:

```
1:  M = ∅
2:  C(n, dummy) // Initialize candidates
3:  m(n, false)    // Initialize matched vertices
4:  while E ≠ ∅ do
5:    for all e = {v, u} ∈ E do
6:      if better_candidate(e, C[v]) then
7:        C[v] = e
8:      if better_candidate(e, C[u]) then
9:        C[u] = e
10:
11:   for all e = {v, u} ∈ E do
12:     if e == C[v] and e == C[u] then
13:       M = M ∪ e
14:       m[v] = true
15:       m[u] = true
16:
17:   for all e = {v, u} ∈ E do
18:     if m[v] or m[u] then
19:       E = E \ {e}
20:     else
21:       C[v] = dummy
22:       C[u] = dummy
23:
24: return M
```

by its two end vertices u and v, the weight of the edge, and a unique ID. The ID is only stored for a simpler way to break ties, especially when dealing with multi graphs. The array C is used to store the heaviest incident edge of each vertex, we call these edges *candidates*. Initially, every entry is set to a dummy edge. The dummy edge has the property that its weight is smaller than the weight of any other edge. The array m stores for each vertex whether it is matched or not, this helps us to remove matched edges and edges adjacent to a matched edge. Every entry is initialized to *false*.

In each round we iterate three times through the remaining edges. In the first iteration (line 5) we *set the maximal incident edge* of each remaining vertex with a degree $\neq 0$. For each edge $e = \{u, v\}$ we check if it is heavier than the heaviest candidate of u. If

that is the case, we update the candidate of u. The same check is done for the other end vertex v. We use the function *better_candidate* to check if e is heavier than the candidate. The function compares the edge weights and, in the case of a tie, it uses hash values of the edge IDs to break the tie. The hash value of an edge ID is computed using the 32Bit or 64Bit integer hash function proposed by Wang in [35]. By using a hash function we hope to achieve a random permutation of the edge IDs and thus avoiding a situation like the one shown in Figure 4.1.1 where we have increasing edge weights along a path. In such a case we would only get a few locally heaviest edges each round, therefore increasing the runtime.

During the second iteration (line 11) of the edges we *identify locally heaviest edges*. An edge $e = \{u, v\}$ is a locally heaviest edge if it is the heaviest incident edge of both end vertices u and v. In this case the edge e is added to the matching M and the end vertices u and v are set to be matched.

The last iteration is used to *remove matched edges* and edges adjacent to matched edges. Those two cases correspond to the case that at least one of the end vertices is matched. If an edge is not removed it is a candidate for a locally heaviest edge during the next round. In this case we set the candidates of the end vertices to the dummy edge again. Otherwise, we could have stored a candidate entry of an edge of heavier weight that has been removed. Although in our case the word 'remove' is technically incorrect. Instead of removing edges we *deactivate* them. This is done by using the first part of the array to store active edges and the second part to store inactive edges. This separation is indicated by a reference to the first inactive entry. Initially all entries are active and the separation reference is set to m. Now whenever we 'remove' an edge we swap it with the last active edge and decrement the separation reference by one, obviously this can be done in $O(1)$. Also in this case we cannot increment the array position in the for-loop (line 17) because we just copied another edge to this position.

The iterations through the remaining edges are done by iterating from entry 0 up to the separation reference.

4.1.2. Theoretical Analysis

Lemma 4.1. *Algorithm 4.1.1 returns a maximal matching M for a graph $G = (V, E)$ with unique edge weights.*

Proof. At first we show that the returned set of edges M is a correct matching, that is, no two edges of M have a common end vertex. Let there be two edges $e, e' \in M$ such that both edges share a common end vertex. Obviously both edges cannot be added in the same round because only one of them can be a locally heaviest edge (they have a common end vertex and edge weights are unique). But when we add one of the two edges to M we remove the other one from the graph G because it is adjacent to the first edge. Thus, there cannot be two adjacent edges in M.

Now we have to show that M is maximal. A matching is not maximal if there are two vertices $u, v \in V$ such that both vertices are not incident to edges of M and there is an edge $e \in E$ which has those two edge as its end vertices. There are only two cases in

which e is removed from G. The first case is that e was matched, which it was not, the other case is that e is adjacent to a matched edge. But both u and v are not incident to matched edges and thus e is not adjacent to a matched edge. Hence e was not removed from the graph and the algorithm would not have terminated because $E \neq \emptyset$.

□

Lemma 4.2. *The weight of the matching M computed by Algorithm 4.1.1 is at least half the weight of the optimal result.*

Proof. Algorithm 4.1.2 is a valid run of Preis' algorithm, only locally heaviest edges are added, the LAM algorithm could decide to do it in the same order. Thus the result must be at least one half the weight of the optimal result, as it has been shown for Preis' algorithm.

□

Lemma 4.3. *The runtime of a single round of Algorithm 4.1.2 is linear in the number of the remaining edges and independent from the number of vertices.*

Proof. Let m be the number of remaining edges at the start of each round. We iterate over each edge 3 times. For most operations it is easy to see that the time spent for a single edge is constant. The problematic operations are the removal of an edge and the check if one edge is heavier than another edge. This check consists of at most two comparisons and the computation of two hash values. The hash functions from [35] consist of a constant number of primitive operations (like a shift). Hence the total runtime of the check is constant.

As described in Section 4.1.1 the removal of an edge can be done in constant time, as well. This gives us a total runtime of $O(m)$ for a single round of Algorithm 4.1.2.

The iteration over the remaining edges of the graph is independent from the number of vertices of the graph, because all remaining edges are stored in one consecutive block. Hence if there are any degree zero vertices those are never considered during a round.

□

The following analysis for the expected runtime of Algorithm 4.1.1 is based on a *slight variation* of the algorithm. For the analysis we assume an algorithm that assigns random edge weights to all remaining edges at the start of each round. This addition does not change the asymptotic runtime of a single round, we just have to iterate one more time over the edges. However the result no longer gives us any approximation guarantees for the maximum weighted matching problem, but the adjusted algorithm still computes *maximal matchings*.

Theorem 4.1. *The expected number of removed edges of Algorithm 4.1.1 during one round is at least half the number of remaining edges, if we assign random weights to each remaining edge at the start of a round.*

Proof. The probability $p(e)$ of a single edge $e = \{u, v\}$ to be a locally heaviest edge is

$$p(e = \{u, v\}) = \frac{1}{d(u) + d(v) - 1}.$$

The edge e must be the heaviest edge of the $d(u)+d(v)-1$ distinct incident edges of the end vertices u and v, and it is equally likely for each of those edges to be the heaviest one (because of the random weights). This gives us

$$s = \sum_{e \in E} p(e)$$

for the expected number of heaviest edges, and thus matched edges, during one round.

To get a lower bound for the expected number of removed edges, we count marks that we assign to removed edges. An edge is removed if it is either a matched edge or incident to a matched edge. For a matched edge $e = \{u, v\}$ we count two marks and for each edge incident to e we count one mark. This gives us a total of $d(u) + d(v)$ marks that are counted for each matched edge. This situation is shown in Figure 4.1.3. Therefore

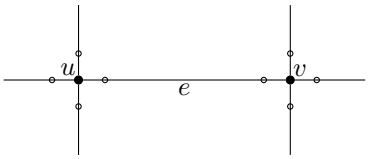


Figure 4.1.3.: Edge with adjacent edges and marks.

the expected number of counted marks of an edge $e = \{u, v\}$ is

$$\frac{d(u) + d(v)}{d(u) + d(v) - 1},$$

i.e. the probability that e is a locally heaviest edges times the number of marks counted for e. Because of the *linearity of expectations* we get for the total expected number of counted marks of all edges:

$$\sum_{e=\{u,v\} \in E} \frac{d(u) + d(v)}{d(u) + d(v) - 1} \geq m$$

Obviously we counted for each matched edge two marks. The other removed edges are incident to *up to* two matched edges, hence we counted up to two marks for those edges and therefore we counted up to *two marks* for each matched edge. This gives us a lower bound of $m/2$ for the expected number of removed edges during a single round.

□

Theorem 4.2. *The expected runtime of the adjusted version of Algorithm 4.1.2, to compute maximal matchings, is in $O(m + n)$. The adjusted algorithm assigns random weights to the remaining edges at the start of each round*

Proof. For the analysis of the runtime of the main while-loop of Algorithm 4.1.2 we distinguish between two events (similar to [27, Theorem 5.8]): The first event is that the number of remaining edges $R(m)$ is less than $\frac{3}{4}m$, we call this event a *good round*.

The other event is the case that at least $\frac{3}{4}m$ of the edges remain, we call this event a *bad round*.

We know that the expected number of removed edges during one round is at least $m/2$ (Theorem 4.1), hence we get for the expected number of remaining edge:

$$E[R(m)] < \frac{m}{2} \tag{4.1}$$

Therefore a round is good as long as the number of remaining edges does not deviate more than 1.5 times from its expected value:

$$\frac{3}{2}E[R(m)] \overset{(4.1)}{<} \frac{3}{4}m$$

Markov's inequality [27, (A.4)] gives us an upper bound for the probability that $R(m)$ deviates by factor of $3/2$ from it is expected value $E[R(m)]$, that is the probability that a round is bad:

$$\text{prob}(R(m) \geq 3/2\ E[R(m)]) \leq \frac{1}{3/2} = \frac{2}{3}$$

Obviously larger deviations are less likely. Hence the *probability that a round is good* is at least $\gamma = 1/3$.

The runtime of a single round is at most cm for some constant c (Lemma 4.3). For the estimation of the runtime $T(m)$ of the main while-loop of Algorithm 4.1.2 we make the conservative assumption that a good round only removes $1/4$ of the remaining edges and a bad round removes no edges at all:

$$\begin{aligned} & T(m) \leq \gamma\ T\left(\frac{3\ m}{4}\right) + (1-\gamma)\ T(m) + cm \\ \Leftrightarrow \quad & \gamma\ T(m) \leq \gamma\ T\left(\frac{3\ m}{4}\right) + cm \\ \Leftrightarrow \quad & T(m) \leq T\left(\frac{3\ m}{4}\right) + \frac{c}{\gamma}m \end{aligned}$$

Inserting $1/3$ for γ gives us an upper bound for the expected runtime $T(m)$ of the main while-loop:

$$T(m) \leq T\left(\frac{3\ m}{4}\right) + 3cm \leq \sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i\ 3cm = 3cm\ \frac{1}{1-3/4} = 12cm = O(m)$$

Before the main while-loop we have a initialization phase, used to allocate memory to store temporary values. The asymptotic runtime of this phase is in $O(n)$. Hence we get for the total expected runtime of $O(m+n)$.

Assuming a graph without any degree 0 vertices, we get that $n \in O(m)$ and the total expected runtime of the algorithm is in $O(m)$.

□

4.1.3. Matchings and Independent Sets – Luby's algorithm

As we have already mentioned Manne and Bisseling point out a relation between Preis' algorithm to compute approximate maximum weighted matchings and an algorithm by Luby to compute *maximum independent sets* [25]. An independent set of a graph $G = (V, E)$ is a subset I of the vertices V, such that no two vertices of I are adjacent in G.

At first we define the notion of a *line graph* which is necessary for the relation between independent sets and matchings. In a line graph $L(G)$ of a graph G each vertex of $L(G)$ corresponds to an edge of E and two vertices of $L(G)$ are adjacent to each other if their corresponding edges in G are adjacent [25]. An example for such a transformation is shown in Figure 4.1.4.

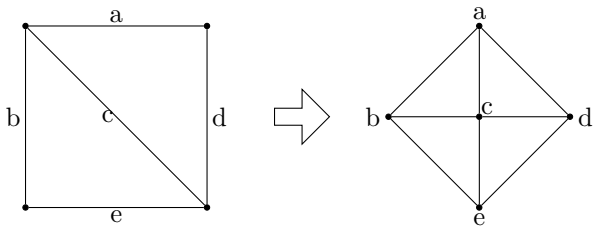


Figure 4.1.4.: Transformation of a graph G (left) to a line graph $L(G)$ (right).

Obviously an independent set of a line graph $L(G)$ of a graph G directly corresponds to a matching of the graph G.

Luby describes in [23] a round based algorithm for computing a maximal independent set of a graph $G = (V, E)$ that corresponds to Algorithm 4.1.1. Luby's algorithm computes a random permutation π of the vertex IDs at the start of each round and then a set I' of vertices such that each vertex of I' is locally minimal, i.e.

$$I' = \{v \in V \mid \forall w \in \text{adj}(v) : \pi(v) < \pi(w)\} \ .$$

This set is then added to the independent set and all vertices of I' are removed from the graph.

The only real difference is that Luby uses IDs instead of weights. We would like to point out that in Algorithm 4.1.1 only the ordering of the weights matters and not the absolute difference between two weights. Hence, one could assign IDs to the edges using the ordering given by the weights, which would basically result in the same algorithm. Luby uses his algorithm to describe a distributed algorithm for the maximal independent set problem as we are going to describe a distributed version of Algorithm 4.1.1.

One could argue that, instead of solving the matching problem directly, we transform a given graph to its corresponding line graph and then solve the independent set problem using Luby's algorithm. The problem is that the transformation might require a lot longer than solving the actual problem. E.g. think of a star, i.e. a connected graph G where each vertex has degree one except for a single vertex with higher degree, each edge is incident to this vertex. Such a graph has $m = n - 1$ edges and its corresponding

line graph $L(G)$ is a complete graph with m vertices. Each edge of G is adjacent to each other edge of G. A complete graph K_m consists of $m(m-1)/2$ edges. Therefore, $L(G)$ is quadratic in the size of the number edges (and vertices) of G.

4.2. Mixed Algorithm – Karp-Sipser and Local Max

In this section we introduce another variation of the Karp-Sipser algorithm to compute approximate maximum cardinality matchings. This algorithm is a mix of the degree 1 reduction of the Karp-Sipser algorithm and the round based approach from Section 4.1. Algorithm 4.2.1 shows the basic structure of this variation. At first we perform a degree 1

Algorithm 4.2.1 Compute an approximate cardinality matching

mixed$(G = (V, E))$:

```
1:  M = ∅
2:  while G ≠ ∅ do
3:    // degree 1 reduction
4:    while ∃v ∈ V : deg(v) = 1 do
5:      pick an arbitrary degree 1 vertex v
6:      e  = incident_edge(v)
7:      M = M ∪ {e}
8:      remove_matched_edge(G, e)
9:    // Match locally heaviest edges
10:   L  = get_locally_heaviest_edges(G)
11:   M = M ∪ L
12:   remove_matched_edges(G, L)
13: return M
```

reduction as long as there are degree one vertices. Afterwards, we add the locally heaviest edge (e.g. based on edge IDs) to the matching. Theses two steps are repeated as long as the graph is not empty.

Obviously the heuristic phase of the algorithm is problematic. Unlike the Karp-Sipser algorithm the heuristic to match each locally heaviest edge does not try to minimize the number of non-optimal decisions. The Karp-Sipser algorithm only makes a single non-optimal decision and then tries to make optimal decisions if possible. But Algorithm 4.2.1 tries to minimize the number of rounds of the outer while-loop. Therefore the algorithm might be suitable for a BSP-style approach to parallelize it. A slight variation would be to match only a fraction of the locally heaviest edges. This might increase the number of rounds, but on the other hand this variation would try to make fewer non-optimal decisions.

4.2.1. Implementation Details

Unlike in the implementation of the local max algorithm, we are using a graph data structure based on an adjacency array [27, Section 8.2]. Such a data structure consists

of two arrays. One array which stores the adjacent edges of a vertex in a block and a second array which stores for each vertex the start position of the vertex's adjacent edges in the edge array. The edge block of vertex 0 starts at position 0, the block of vertex 1 starts right after the block of vertex 0 and so on. To get the end of one edge block, one just has to look for the start of the next block. We are using a dummy vertex to make sure that this invariant also holds for the last vertex. A simple example for such a graph representation is shown in Figure 4.2.5.

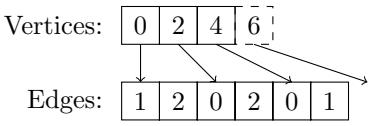


Figure 4.2.5.: Adjacency array example for a triangle graph.

We also must be able to *delete edges*. Obviously, a traditional adjacency array is not suitable for such a situation. Therefore, we are using the same mechanism as for the edge graph from Section 4.1. Instead of deleting edges, we simply have two parts for the blocks of incident edges of a vertices. The first part manages active edges and the second part is for inactive edges. When deleting/deactivating an edge, we just have to swap the edge with the last active edge. Within the vertex array we store the start of each block and the end of the block of active edges. This also makes it easy to compute the actual degree of a vertex v and the *active degree* of v, i.e. the number of active incident edges of v.

As before we would like to be able to iterate over just the active edges. But in this structure we do not have any continuous block of active edges and iterating over the vertices and their incident active edges would cause at least $O(n)$ steps per round, because we would iterate degree zero vertices, too. Instead, we are using a list to keep track of *active vertices*, those are the vertices with a degree of at least one. We also use another list of vertices to keep track of *degree one vertices*.

For the *degree 1 reduction phase* we just have to pick a vertex from the list of degree one vertices, add its incident edge to the matching and remove it from the graph. The most complex operation is the removal of an edge. We will get to that shortly.

During the second phase we can implement the operation to pick *locally heaviest edges* in a similar way as we do it in Algorithm 4.1.2. We just have to iterate over active vertices and their incident active edges. Again the most complicated operation is the removal of an edge. It is not enough to just set it inactive in the graph structure (as we have already described). Deleting an edge also changes the degree of vertices. They might become degree one vertices in which case we have to add them to the list of degree one vertices or, in the case that a vertex becomes a degree zero vertex, we have to remove it from the corresponding list. So far we are only deleting matched edges, but this means that we also have to delete their adjacent edges. The implementation of the removal of an edge $e = \{u, v\}$ from the graph G is shown in Algorithm 4.2.2. We use a list to store all modified vertices, i.e. we have deleted an adjacent edge. This list is used to check for new degree one or degree zero vertices.

Algorithm 4.2.2 Delete an edge

delete_edge($e = \{u, v\}$, $G = (V, E)$):

```
1:  modified_vertices = ∅
2:  modified_vertices = modified_vertices ∪ u
3:  modified_vertices = modified_vertices ∪ v
4:  G = G \ {e}
5:
6:  for all active incident edges e' = {u, v'} of u do
7:      modified_vertices = modified_vertices ∪ v'
8:      G = G \ {e'}
9:
10: for all active incident edges e' = {u', v} of v do
11:     modified_vertices = modified_vertices ∪ u'
12:     G = G \ {e'}
13:
14: for all v ∈ modified_vertices do
15:     if active_degree(v) == 1 then
16:         add v to list of degree one vertices
17:     else if active_degree(v) == 0 and v in list of degree one vertices then
18:         remove v from list of degree one vertices
```

At first we add the end vertices u and v of the edge e to the list of modified vertices, and afterwards we remove e from the graph. In the next two for-loops we add the missing end vertices of the adjacent edges of e to the list of modified vertices and also remove those edges from the graph. Removing/deactivating an edge from the graph G can be done in constant time as we have already explained. Hence the total time for those two loops is in $O(\deg(u) + \deg(v))$ and the number of modified vertices is $\deg(u) + \deg(v)$. The degrees correspond to those at the start of the delete operation.

The last loop checks whether each *modified vertex* v is a degree one vertex or a degree zero vertex. In the case of a degree one vertex we add the node to the list of degree one vertices. If the degree of v is zero (line 17), we check if v is in the list of degree one vertices and remove it if necessary. This check and removal can be done in constant time by using an extra array to store the position of each vertex within the list of degree one vertices and a move operation that copies the last vertex of this list to the position of the vertex that is removed. This is similar to our swap operation. Without this check it would be possible that degree zero vertices remain in the list of degree one vertices. For example during the heuristic phase we do not delete all edges at once, we delete them one after another. Thus, it is possible that a vertex temporarily becomes a degree one vertex. This gives us a total runtime of $O(\deg(u) + \deg(v))$ for a single deletion operation. An implementation without this check for degree zero vertices would require an extra degree check during the degree 1 reduction phase and we would need a deletion operation for all locally heaviest edges during the second phase. Only one thing is missing, we have not removed degree zero vertices from the list of active vertices. This is done by a single

run through this list and checking for degree zero vertices.

Let M_1 be the set of matched edges of the degree 1 reduction phase. The set M_1 has at most $n/2$ edges, i.e. each vertex is matched. As we have already shown the removal of an edge $e = \{u, v\}$ has a maximal runtime of $c(d(u) + d(v))$, with c a constant. This gives us an upper bound for the runtime of the degree 1 reduction phase of

$$\sum_{e=\{u,v\}\in M_1} c(\deg(u) + \deg(v)) \leq \sum_{v\in V} c \ \deg(v) = 2cm = O(m) \ .$$

Finding locally heaviest edges also has a runtime of $O(m)$ and the runtime of deleting locally heaviest edges is in $O(m)$ with the same argument used for the degree 1 reduction phase. Removing degree zero vertices from the list of active vertices is not more expensive than finding locally heaviest edges and thus the runtime is in $O(m)$ of this operation. In total this gives us a runtime of a single round of Algorithm 4.2.1 of $O(m)$, where m is the number of remaining edges at the start of a round.

4.3. Local Tree Algorithm

In this section we introduce a new algorithm to compute approximate maximum weighted matchings, it is based on the computation of maximum weighted matchings of trees (Algorithm 3.3.1). The algorithm also has some similarities with the GPA algorithm introduced in Section 3.4 and the local max algorithm from Section 4.1 (Algorithm 4.1.1). Like the local max algorithm, it works in rounds. At the start of each round we compute a subset L of the remaining edges. This time however, we select edges if they are the heaviest edge at one of their end vertices (not necessarily at both vertices) and add them to the set L. The set L defines a set of trees, see Lemma 4.4. Hence the name *local tree algorithm*. We then compute a maximum weighted matching for each tree defined by L, by dynamic programming, and add those matchings to the final matching computed by the algorithm. We also remove each matched edge and the adjacent edges from the given graph. The algorithm terminates as soon as there are no more edges left. Those steps are similar to the GPA algorithm. In both cases we compute a set of simple subgraphs at first and then compute optimal results for those subgraphs.

Algorithm 4.3.1 Compute an approximate weighted matching

local_tree_matching$(G = (V, E))$:

1: $M = \emptyset$
2: **while** $E \neq \emptyset$ **do**
3: $L =$ *heaviest_incident_edges*(G)
4: $F =$ *forest*(L)
5: $M' =$ *maximum_weighted_matching_forest*(F)
6: $M = M \cup M'$
7: *remove_matched_edges*$(G,\ M')$
8: **return** M

The just described local tree algorithm is shown in Algorithm 4.3.1. The procedure *heaviest_incident_edges* computes for each vertex of the given graph the *heaviest incident edge* and returns those edges. Computing a maximum weighted matching for a set of trees corresponds to the computation of a maximum weighted matching of a forest.

Figure 4.3.6 shows an example of a run of the local tree algorithm. In this example the local tree algorithm performs only a single round. The regular and dashed lines are

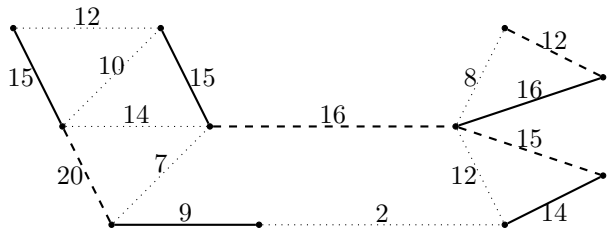


Figure 4.3.6.: Example for local tree algorithm.

the edges of the computed trees. Dotted lines represent edges which are not the heaviest incident edge at one of their end vertices. As one can see there are two trees in this example. The regular lines specify the edges of the maximum weighted matchings of the two trees. After adding those edges to the resulting matching and removing them from the graph no more edges are left and the algorithm terminates after the first round.

4.3.1. Implementation Details

For the implementation of the local tree algorithm we chose to use the same graph data structure as in Section 4.1, i.e. the graph is represented by a simple array containing all the edges of the graph. As described before, edges can be set inactive by swapping it with the last active edge.

Computing the heaviest incident edge of each vertex is done in a similar way as the computation of locally heaviest edges in Algorithm 4.1.2. We use an array to store a reference to the heaviest incident edge for each vertex. Initially, each entry is set to a dummy edge, which is lighter than any edge of the graph. In the actual implementation, the memory for this array is allocated once outside of the function *heaviest_incident_edge*, not like it is shown in Algorithm 4.3.2.

The heaviest incident edges of vertices are identified by one iteration through the remaining edges and a second iteration through those edges is used to add the heaviest incident edges to the set L.

The forest defined by the set L of heaviest incident edges is represented by an adjacency list. We chose this structure because it supports fast insertion and deletion of edges, and thus allows us to create a forest of L in linear time in the size of L. The forest structure F initially allocates memory for all n vertices. Where n is the number of vertices of the graph G passed to the local tree algorithm. This does not influence the asymptotic runtime of the algorithm. During the first round of the algorithm we need memory for each vertex, because for each vertex v there is at least one edge in L which has v as an end vertex.

Algorithm 4.3.2 Return the set of heaviest incident edges

heaviest_incident_edges($G = (V, E)$):

```
1:  L = ∅
2:  C(n, dummy) // Initialize candidates
3:  for all e = {v, u} ∈ E do
4:    if better_candidate(e, C[v]) then
5:      C[v] = e
6:    if better_candidate(e, C[u]) then
7:      C[u] = e
8:
9:  for all e = {v, u} ∈ E do
10:   if e == C[v] or e == C[u] then
11:     L = L ∪ e
12:
13: return L
```

Now we can describe how we create the forest F defined by L. At first we add each edge $e = \{u, v\}$ of L to F, e is added to the adjacency lists of u and v. Obviously, the insertion of edges can be done in time $O(|L|)$. During the next step we actually build the forest, i.e. we decide on a root vertex for each tree of F and remove backward edges.

Algorithm 4.3.3 Build a forest

build_forest(F):

```
1: roots = ∅
2: for all edge e = {v, u} of F do
3:   if v not visited then
4:     roots = roots ∪ v
5:     build_tree(v, F)
6: return roots
```

The *build_forest* method (Algorithm 4.3.3) iterates over each edge e of F and checks if the first end vertex v of e has been visited. We have found a new tree if v has not been visited. In this case we add v to the list of roots and build the tree starting at v. This is done by iterating over incident edges. In case of a forward edge, we make a recursive call to *build_tree*, otherwise we remove the backward edge. We set all vertices, visited during this recursive procedure, to the state visited. While building the forest we visit each edge three times, once during the *build_forest* procedure and twice during the *build_tree* method (backward and forward edges). This gives a asymptotic runtime of $O(|L|)$ to build the forest.

The next step is to compute a maximum weighted matching M of the forest F, as shown in Algorithm 4.3.4. The computation of M is based on the computation of maximum weighted matchings of trees, as described in Section 3.3. We will not go into

Algorithm 4.3.4 Build a tree starting at a given vertex

maximum_weighted_matching_forest(F):

1: $roots = roots\_of\_forest(F)$
2: $M = \emptyset$
3: **for all** $r \in roots$ **do**
4: $\quad M = M \cup weighted\_matching\_of\_tree(r, F)$
5: **return** M

detail on how to implement the algorithm to compute maximum weighted matchings of trees, as we have already done this (Algorithm 3.3.1), but we will talk about the modifications that have been necessary to compute the matching in case of a forest. The first difference is that trees are no longer specified by a tree data structure, instead they are specified by the root vertex and the forest F. The next and biggest difference is the usage of the outgoing_edge-array, which specifies for each vertex of the tree which outgoing edge is used for the matching, if the incoming edge is not matched, and the subtree_result-array, which stores the two possible weights of matchings for each subtree. In the case of the tree-matching algorithm those arrays have been allocated for a single tree, but in our case those arrays are allocated once at the start of the algorithm and have one entry for each vertex of the provided graph G. This does not influence the asymptotic runtime of the local tree algorithm. The required time is in $O(|L|)$, because $|L| \geq n/2$. Apart from those two changes there are no differences compared to the algorithm from Section 3.3. Using those arrays does not change the runtime of the tree matching algorithm. This gives us a total runtime for the forest matching algorithm of:

$$\begin{aligned}\sum_{r \in roots} \text{runtime}(\text{tree}(r)) &\overset{\text{sec 3.3}}{\leq} \sum_{r \in roots} c \ \text{size\_of\_tree}(r) \\ &= c \sum_{r \in roots} \text{size\_of\_tree}(r) = O(\text{size\_of\_forest})\end{aligned}$$

In the last phase of the local tree algorithm we add the matched edges to the matching and set their end vertices to the state matched. Afterwards, we iterate over all remaining edges and remove them if they are incident to a matched vertex. This is the same procedure as used in Section 4.1 to add matched edges to the resulting matching and to remove edges incident to a matched vertex. The runtime of this is $\Theta(m)$, where m is the number of remaining edges.

4.3.2. Theoretical Analysis

Lemma 4.4. *The set L of heaviest incident edges computed during a round of the local tree algorithm, defines a set of trees (a forest).*

Proof. Obviously any set of edges defines a set of connected graphs, each connected graph consists of at least one edge. It now remains to show that each of those connected graphs is a tree.

A tree is a connected graph without any cycles, thus a connected graph is not a tree if it contains at least one cycle.

Now let us assume that L does not define a set of trees, thus there must be a subset C of edges in L defining a cycle. This cycle C must have a unique *lightest edge* e (because of the tie breaking), but this edge cannot be the heaviest incident edge of one of its end vertices. Both end vertices of e are incident to other edges with a higher weight. Therefore e cannot be contained in C and C cannot build a cycle.

□

Lemma 4.5. *The set of edges M computed by Algorithm 4.3.1 is a maximal matching.*

Proof. Obviously edges from the matchings of the local trees cannot be adjacent to each other, because they are not connected by edges. Edges which remain at the end of one round are not adjacent to any matched edge. Hence the resulting matching must be correct, because it is the union of matchings of local trees.

The matching M is maximal because only matched edges and edges adjacent to them are removed by Algorithm 4.3.1 and the algorithm only terminates if no edges remain.

□

Theorem 4.3. *The solution computed by Algorithm 4.3.1 might be an arbitrarily bad approximation of the optimal result.*

Proof. Consider the example from Figure 4.3.7. The first subfigure shows the input graph with weights assigned to the edges. With the conditions that $\varepsilon > 0$ and $a > \varepsilon$ we get Subfigure 4.3.7(b) as the temporary tree computed by the local tree algorithm. This subfigure shows the maximum weighted matching of the tree (dashed edges are unmatched). Hence, the weight computed by the local tree algorithm would be $a + 3.5\varepsilon$. But the optimal result for small ε (ε a lot smaller than a) is shown in Subfigure 4.3.7(c). The weight of the optimal result is $3a - 2.5\varepsilon$. For $\varepsilon \to 0$ we get that the result computed by the local tree algorithm is about a and the optimal result would be $3a$. We can easily extend the graph by more subgraphs of the form shown in Figure 4.3.8 at the vertex v. Each such subgraph only adds the weight 2ε to the weight computed by the local tree algorithm but it adds the weight $a - \varepsilon$ to the optimal result.

Therefore, the approximation computed by the local tree algorithm might be arbitrarily bad.

□

Lemma 4.6. *Each local tree has a unique heaviest edge e. Let one of the end vertices of e be the root of the tree. All paths from the root to a leaf vertex have* decreasing edge weights*. In case of unique edge weights or tie breaking the weights are strictly decreasing (using the tie breaking oder).*

Proof. Let us assume there is a path from the root r to a leaf l without decreasing edge weights. In this case there must be at least one edge g such that the subsequent edge h is of higher weight. Further, let us assume that g is the first edge with this property. Obviously, g cannot be the root edge e, because this edge is the heaviest edge of the

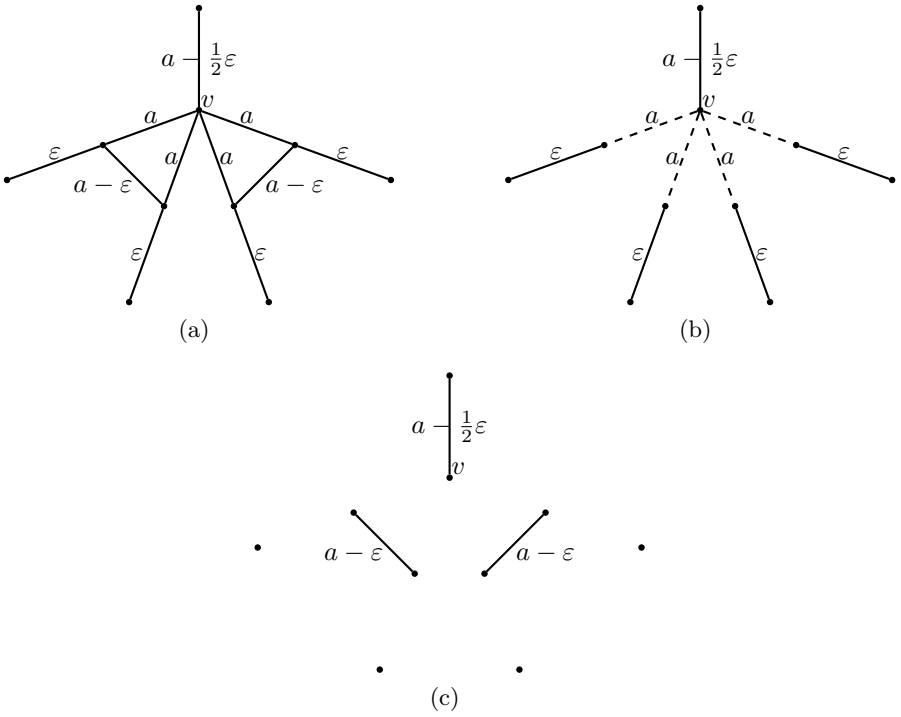


Figure 4.3.7.: Bad approximation computed by the local tree algorithm. Figure (a) shows the graph, figure (b) shows the temporary tree computed by the algorithm and the resulting matching and figure (c) shows the optimal solution for small ε.

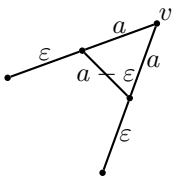


Figure 4.3.8.: Subgraph of the graph shown in Figure 4.3.7(a).

tree. Hence g must have a predecessor edge f of greater weight. This situation is shown in Figure 4.3.9.

Thus, we get $\omega(f) > \omega(g)$ and $\omega(h) > \omega(g)$ and therefore g cannot be the heaviest incident edge of one of its end vertices u and v. Hence, an edge with the properties of g cannot exist.

The proof also works when r is the other end vertex of e. Using tie breaking we obviously get strictly decreasing edge weights.

□

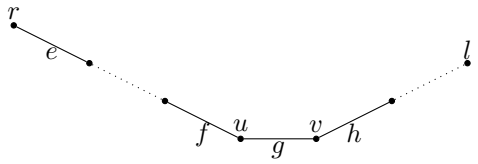


Figure 4.3.9.: Example for decresing paths.

Lemma 4.7. *The runtime of a single round of Algorithm 4.3.1 is* $\Theta(m)$ *and* m *is the number of remaining edges.*

Proof. We have already shown in Section 4.3.1 that the computation of the heaviest incident edges can be done in $O(m)$ and the computations of the forest and the corresponding maximum weighted matching can be done in $O(|L|)$. Finally removing matched edges can be done in $O(m)$ as well. This results in a total asymptotic runtime of $O(m)$ ($|L| \leq m$). Obviously a lower bound is $\Omega(m)$, because we visit each edge at least once.

□

Lemma 4.8. *The maximum number of rounds of Algorithm 4.3.1 is* $\min\{d_{max}, \lfloor n/2 \rfloor\}$ *for graphs without loops, where* d_{max} *is the maximum degree of a vertex of the input graph.*

Proof. Each round we add at least one edge to the matching. As long as there are remaining edges we compute a non empty forest, and the maximum weighted matching of a non empty tree is not empty. Therefore, we match at least two vertices per round, an all of their incident edges are removed. A single unmatched remaining vertex must have degree 0 and hence there are no more remaining edges and the algorithm terminates. This gives us an upper bound for the number of rounds of $\lfloor n/2 \rfloor$.

On the other hand, each edge of the forest computed during a round will be removed at the end of the round, because it is either a matched edge or adjacent to a matched edge. Otherwise, the resulting matching of the forest would not be optimal. For each remaining vertex with a degree $\neq 0$ there is at least one incident edge which is part of the forest, and hence at least one incident edge of each vertex is removed during a round. Therefore, we have at most $d_{\max}$ rounds until each vertex has degree 0 and the algorithms terminates.

□

Lemma 4.9. *An upper bound for the runtime of Algorithm 4.3.1 is* $O(n\ m)$ *(for graphs without degree 0 vertices).*

Proof. According to Lemma 4.8 the maximum number of rounds is $\min\{d_{\max}, \lfloor n/2 \rfloor\} \leq n$ and each round has a runtime of $O(m)$. (Lemma 4.7) This gives us an upper bound of $O(n\ m)$ for the runtime of the local tree algorithm.

□

For an example with a runtime of $\Omega(nm)$, consider a complete graph with vertices $\{1, \ldots, n\}$ and the following weight function:

$$\omega(e = \{u, v\}) = \begin{cases} \max\{u, v\} & , |u - v| = 1 \\ \max\{u, v\} - \varepsilon & , \text{else } (0 < \varepsilon < 1) \end{cases}$$

Figure 4.3.10 shows such a graph for 5 vertices.

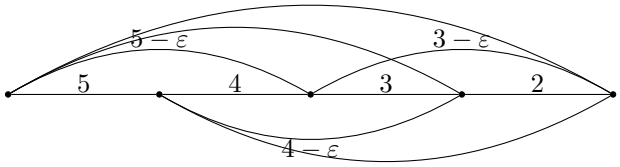


Figure 4.3.10.: Example for local tree algorithm with runtime $O(m^{3/2})$.

During the first round all the incident edges of the vertex n (vertex with the highest ID) are the edges of the forest (which build a star). Obviously, the edge $\{n, n-1\}$ will be matched, because it is the heaviest edge of the resulting star. The resulting graph after removing this edge and the adjacent edges has $n-2$ vertices and the same properties has the initial graph. The lower (and upper) bound of a single round is linear in the number of remaining edges. This gives us a lower bound for the runtime t of Algorithm 4.3.1 for this kind of graph of:

$$t \geq \sum_{i=1}^{n/2} c\frac{2i(2i-1)}{2} = c\sum_{i=1}^{n/2} i(2i-1) = c\left(2\sum_{i=1}^{n/2} i^2 - \sum_{i=1}^{n/2} i\right)$$
$$= c\left(\frac{n(n/2+1)(n+1)}{6} - \frac{n^2+n}{2}\right) = \Omega(n^3) \overset{n\in\Theta(\sqrt{m})}{=} \Omega(nm)$$

Lemma 4.10. *Algorithm 4.3.1 matches each vertex during a round that would be matched by Algorithm 4.1.1, in the case of unique edge weights.*

Proof. Obviously each locally heaviest edge $e = \{u, v\}$ is an edge of a local tree and it also must be the *heaviest edge* of this local tree, otherwise it would not be a locally heaviest edge. Algorithm 4.1.1 only matches locally heaviest edges during a round and therefore only vertices which are incident to a locally heaviest edge.

There are two possible cases if Algorithm 4.1.1 does not match each incident vertex of locally heaviest edges.

The first case is that there is an unmatched locally heaviest edge $e = \{u, v\}$ and both end vertices of e are not matched, but in this case the result will not be optimal. The other case is if there is an unmatched locally heaviest edge $e = \{u, v\}$, but only one of its end vertices is matched. Without loss of generality let the matched vertex be the end vertex u. Then there must be an edge $f = \{u, w\}$ that is adjacent to e which is

matched. But e is heavier than f (e is the heaviest edge of the corresponding local tree) and therefore the weighted matching computed for the local tree would not be optimal.

Thus either e is matched or it is adjacent to two matched edges and therefore both end vertices of e are matched.

□

It can also be shown that the local tree algorithm matches the same vertices as the local max algorithm during a round for non unique edge weights. But we have to adjust our algorithm for that. At first the heaviest edges (according to the tie breaking) of each local tree must be incident to the root of the local tree and if one of the end vertices of a heaviest edge is a leaf then it must be the root of the local tree.

For the analysis of the expected runtime of Algorithm 4.3.1 we again assume that the algorithm assigns random weights to the edges at the start of each round.

Theorem 4.4. *The expected runtime of Algorithm 4.3.1 is in $O(m + n)$, if we assign random unique weights to the remaining edges at the start of each round.*

Proof. Algorithm 4.3.1 removes during each round at least as many edges as Algorithm 4.1.1 would remove, because it matches the same vertices (Lemma 4.10) and both algorithms only remove edges incident to matched vertices. Hence the expected number of removed edges of Algorithm 4.3.1 is at least $m/2$ (Theorem 4.1).

We have already seen that the runtime of a single round of Algorithm 4.3.1 is linear in the number of remaining edges (Lemma 4.7), hence at most cm steps are required for a single round (for a constant c). Using this information and the fact that the expected number of removed edges is at least $m/2$ we can use the same proof used for Theorem 4.2 to show that the expected runtime of the main loop is in $O(m)$.

There is also an initialization phase before the main loop which allocates memory for a constant number of arrays. None of those arrays contain more elements than there are vertices. Hence the total expected runtime is $O(m + n)$.

□

4.4. Experimental Results

In this section we present the experimental results for the three algorithms discussed in Chapter 4 and compare them with each other and with the Karp-Sipser algorithm presented in Section 3.2 and the GPA algorithm from Section 3.4. We only compare algorithms with each other if it is appropriate. The Karp-Sipser and the mixed algorithm try to compute maximum cardinality matchings, hence it does not make much sense to use them to compute maximum weighted matchings. In case of the expansionstar2 rating (introduced later in this section) the results of the mixed algorithm are on average 13% worse than the results of the local max algorithm. On the other hand, algorithms which compute approximate maximum weighted matching can be easily adjusted to compute cardinality matchings by assigning constant weights to the edges.

All algorithms were compiled using version 4.4.3 of gcc and the optimization level was set to -O3. We used our own implementations of all algorithms except for GPA. Especially the runtimes of Karp-Sipser and the mixed algorithm should be taken cautiously. The implementations have the asymptotic runtimes as described before, but there is probably still room for optimization.

To execute the algorithms we used a computer with four Quad-Core AMD Opteron 8350 Processors (2 GHz and 512 KB L2-Cache per core and 2 MB L3-Cache per processor) and 64GB of ram. The operating system was Ubuntu 10.04. We used Valgrind [2] to simulate and count cache misses.

For the experiments we used the Random Geometrics Graphs, Delaunay Graphs and Walshaw's Graphs [33] from the 10th DIMACS Implementation Challenge [5]. We also used coarsened versions of these graphs as described in the introduction and computed by the KaFFPa graph partitioner from [31]. Level 0 graphs are the initial graphs before any coarsening step, level 1 graphs after oner step and so on. The level 0 graphs all have vertex and edge weights of 1. The coarsened graphs might have different vertex and edges weights, as described in the introduction. This results in 556 different graphs.

We also used grid graphs and complete graphs (K_n-graphs) for the computation. Grid graphs represent non periodic grids of dimension d with a length of l vertices in each direction. A K_n-graph is a complete graph with n vertices (i.e. one edge between each pair of vertices).

We executed each of the algorithms, except for GPA, ten times on all graph instances. The GPA algorithm was only repeated four times.

In the experiments we used *edge ratings* for the weights of the edges. To compute maximum cardinality matchings we used constant edge weights. For maximum weighted matchings we used edge weights provided by the input graphs, the expansionstar2 rating from [20] which is

$$expansion^{*2}(e = \{u, v\}) = \frac{\omega(e)^2}{c(u)\ c(v)}$$

and random edge weights from the interval $[0, 1)$.

4.4.1. Runtime and Quality Comparison

In this section we compare the runtimes of different algorithms and the quality of the resulting matchings using the graphs from the 10th DIMACS Implementation Challenge [5] and their coarsened versions. We only consider cardinality matchings and weighted matchings using the expansionstar2 rating. The results of the two other ratings show the same behaviour.

We only compare two algorithms at a time. The results for one graph are simply divided by each other and then a histogram of all results is shown. We decided to use this kind of comparison because of the different structures of the used graphs. The intervals of the histograms are all left inclusive and the dashed line shows the average value.

Cardinality Matching

Figure 4.4.11 shows the comparison of Karp-Sipser and the local max algorithm computing approximate cardinality matchings. In all cases, Karp-Sipser computed matching

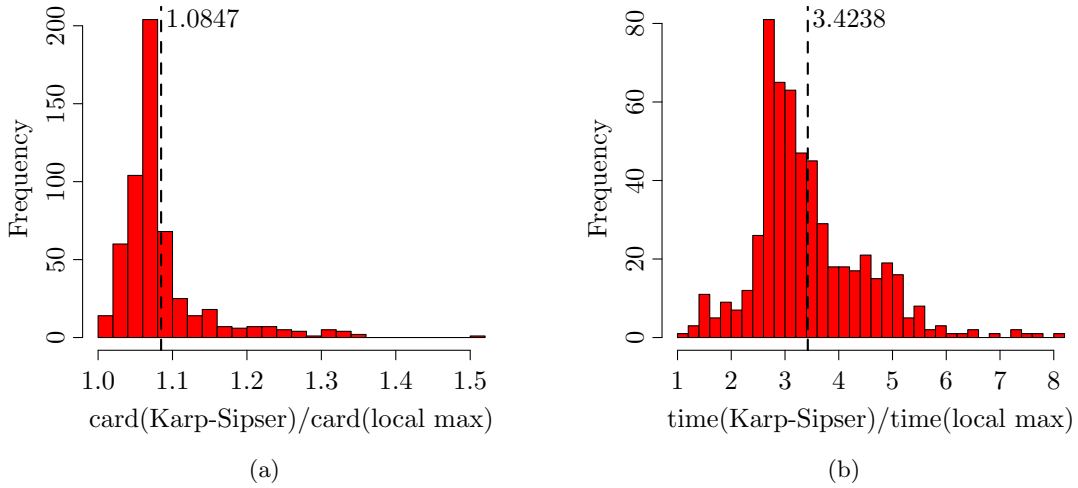


Figure 4.4.11.: Comparison between Karp-Sipser and the local max algorithm. Results are approximate maximum cardinality matchings. The left histogram shows the cardinality comparison and the right one shows the runtime comparison.

of at least the size of the matchings computed by the local max algorithm. On average, the matchings computed by *Karp-Sipser* were about 8.5% *larger*. On the other hand, the *local max algorithm* was, on average, *3.4 times faster* than Karp-Sipser.

The comparison of Karp-Sipser and the local tree algorithm (Figure 4.4.12) shows a similar behaviour. The matchings of *Karp-Sipser* are *larger* than the ones computed by the local tree algorithm (in average 4.7%) and the *local tree algorithm is faster* than Karp-Siper, although not by much. Local tree computes a larger matching for the bcsstk29 graph level 0, bbcsstk33 graph level 6 and the data graph level 4. All of these graphs come from the Walshaw's graph partitioning archive [33].

Again, the cardinality of the matchings computed by Karp-Sipser is in most cases larger than the one computed by the mixed algorithm (Figure 4.4.13). But in this case Karp-Sipser is faster. Those results are not that surprising, since the mixed algorithm is a variation of the Karp-Sipser algorithm. They differ when non-optimal choices are made. Karp-Sipser tries to make as few non-optimal choices as possible. On the other hand, the mixed algorithm does not try to minimize the number of non-optimal choices.

Figure 4.4.14 shows the results of the comparison between the local tree and local max algorithms. In most cases (540 of 556) the *local tree algorithm computes larger matchings* than the local max algorithm, but the *local max algorithm is about* 3.2 *times faster*.

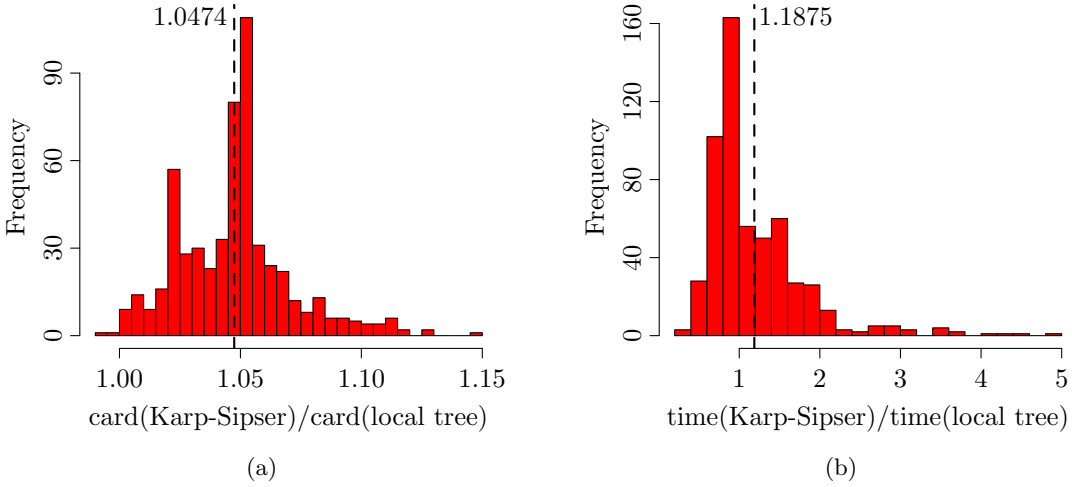


Figure 4.4.12.: Comparison between Karp-Sipser and the local tree algorithm. Results are approximate maximum cardinality matchings. The left histogram shows the cardinality comparison and the right one shows the runtime comparison.

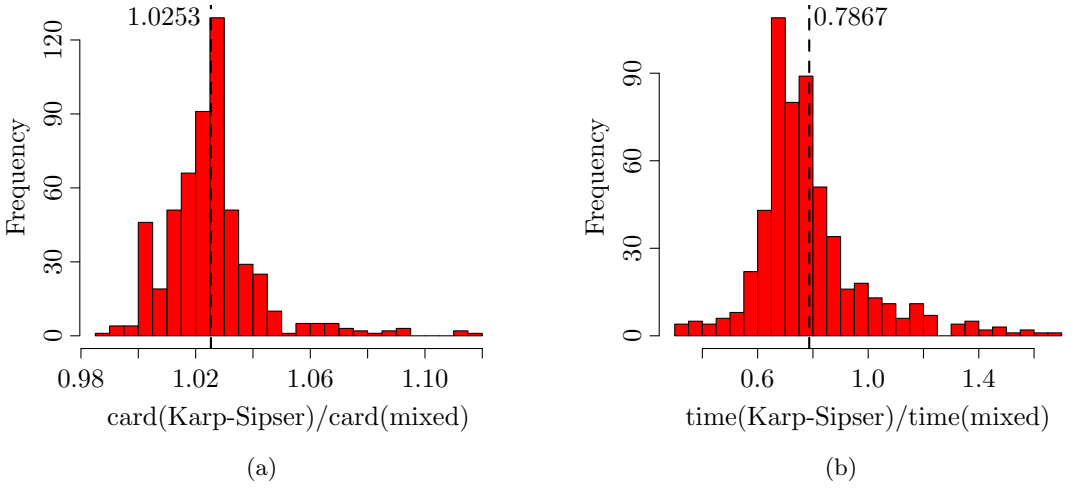


Figure 4.4.13.: Comparison between Karp-Sipser and the mixed algorithm. Results are approximate maximum cardinality matchings. The left histogram shows the cardinality comparison and the right one shows the runtime comparison.

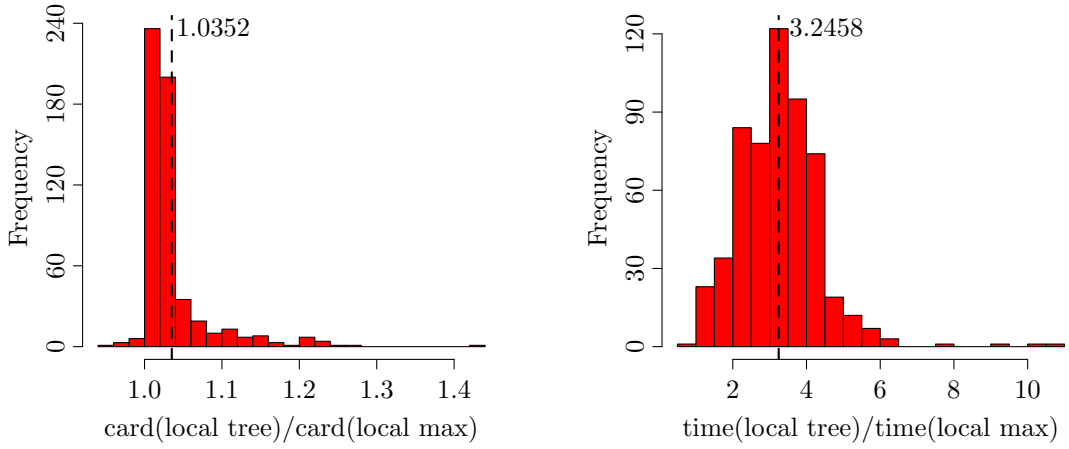


Figure 4.4.14.: Comparison between the local tree and local max algorithms. Results are approximate maximum cardinality matchings. The left histogram shows the cardinality comparison and the right one shows the runtime comparison.

Expansiostar2 Matchings

In the case of the expansionstar2 rating we only compare GPA, the local max algorithm and the local tree algorithm.

Figure 4.4.15 shows the comparison between the local tree algorithm and the local max algorithm. The result is pretty similar to the result of the comparison for cardinality matchings. *Local tree is still better than local max* in most cases (518 of 556) but the overall difference is not as large. The accumulated ratings of the local tree algorithm are only about 0.9% better than the once from local max. An again *local max is about* 3.4 *times faster* than local tree.

More interesting is the comparison of the local tree algorithm and GPA. On average, *both algorithms produce matchings with almost the same accumulated rating.* But still, in 383 of 556 cases the local tree algorithms produces matchings with a larger accumulated rating. Also the *local tree algorithm is about 4 times faster* than GPA.

The comparison between GPA and local max is similar to the one between local tree and local max. GPA computes matchings that are a *little bit better* but on the other hand is a lot *slower than local max* (about a factor of 15).

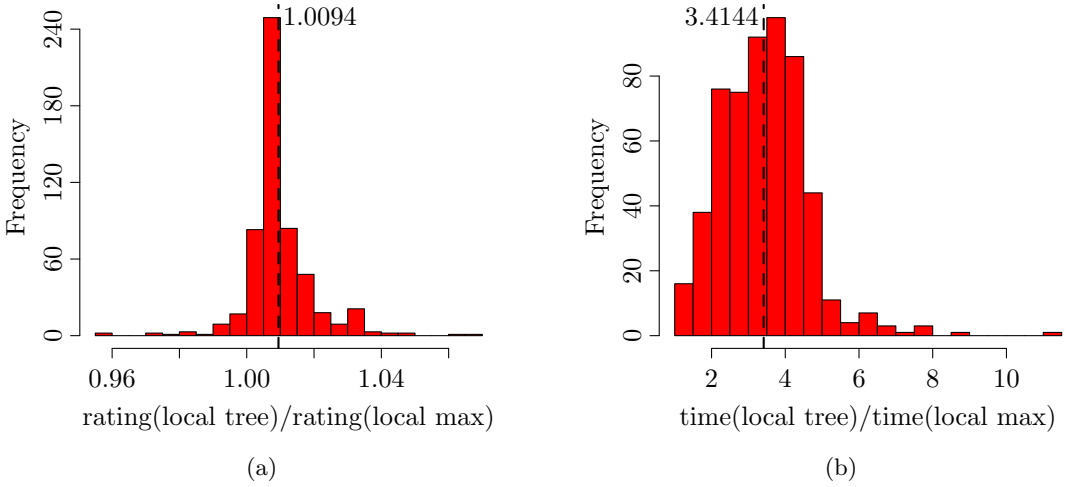


Figure 4.4.15.: Comparison between the local tree and local max algorithms. Using expansionstar2 rating. The left histogram shows the accumulated rating comparison and the right one shows the runtime comparison.

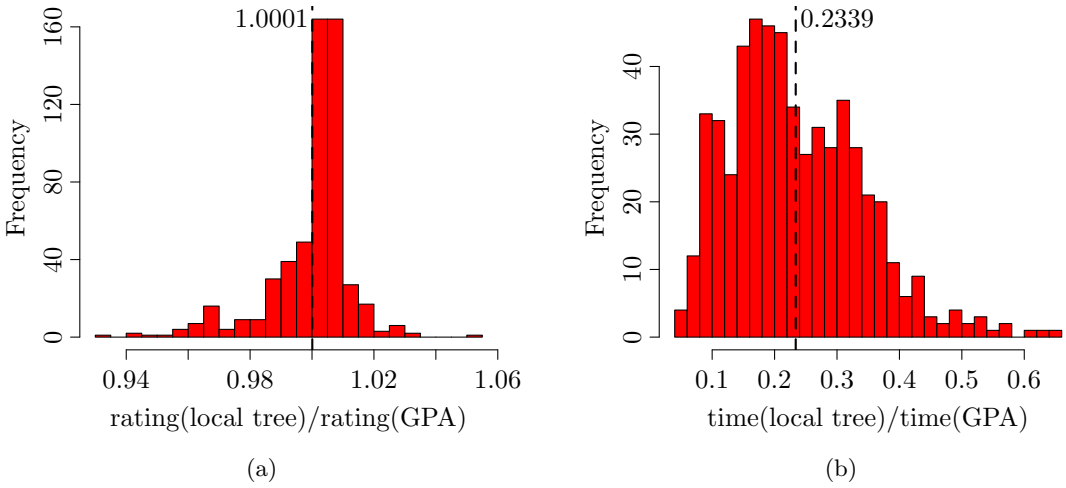


Figure 4.4.16.: Comparison between the local tree algorithm and GPA. Using expansionstar2 rating. The left histogram shows the accumulated rating comparison and the right one shows the runtime comparison.

4.4.2. Edge Development

We have shown in Section 4 that we expect to remove at least half of the remaining edges during each round, for slight variations of the local max and local tree algorithm. In this section we have a look how the number of remaining edges develops in practice.

This time expansionstar2 and random edge weights are used for the edge ratings. And again only graphs from the 10th DIMACS Implementation Challenge and their coarsened versions are considered.

We bundled the edge developments for graphs with the same number of rounds to compute a matching. Because we think that the behaviour for graphs which require the same number of rounds should be roughly the same otherwise the ranges of remaining edges would be too large.

Figure 4.4.17 and Figure 4.4.18 show the edge developments for the local max algorithm. Each figure shows the development for results with the same number of rounds. The number of rounds observed for the local max algorithm ranged from 2 to 10, but we only show results for 4, 5, 6, and 7 rounds, because they represent the *majority of the results*. In case of random edge weights they represent 487 of 556 results and in case of expansionstar2 rating 489 of 556. But the missing results show the same behaviour.

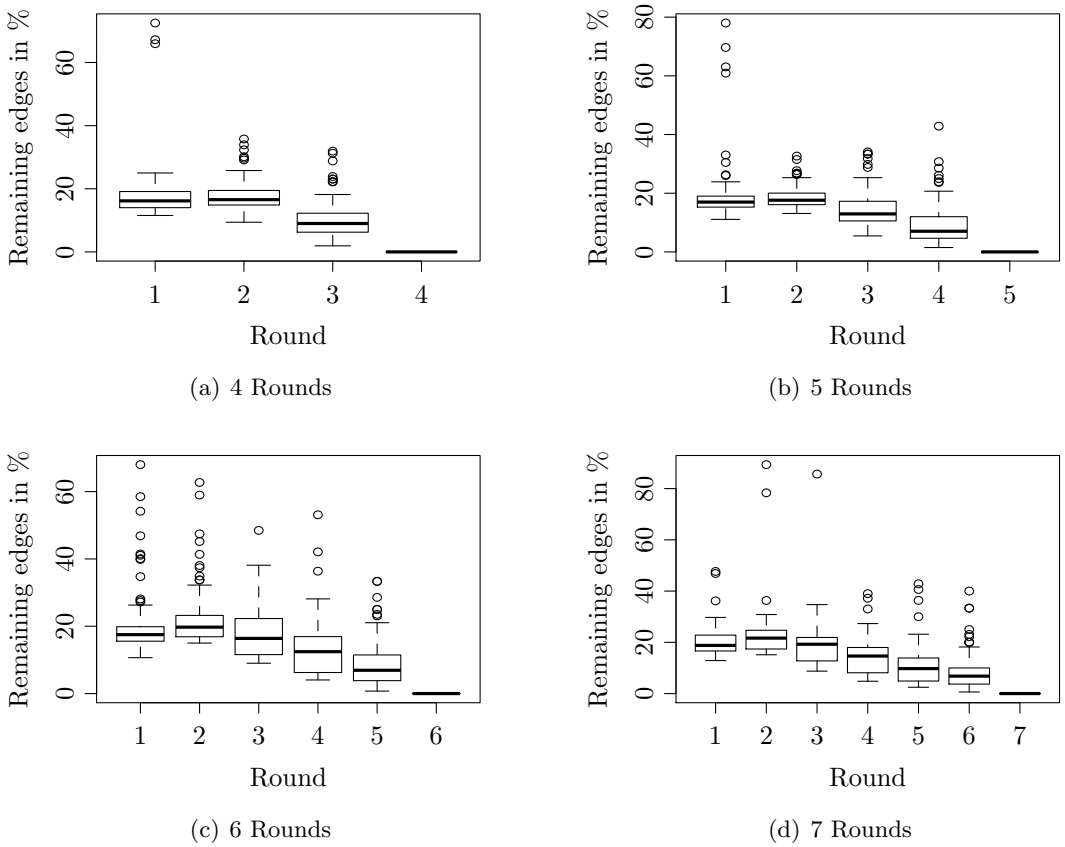
Each entry of the box plots represents the fraction of remaining edges at the end of a round compared to the start of the round. The lower and upper ends of the boxes represent the lower and upper quartiles for the results of a particular round. The thick bar inside the box represents the median. The whiskers represent the lowest and highest values which are within 1.5 times the interquartile range. And the most extreme outliers are represented by circles.

Figure 4.4.17 shows the results when using the expansionstar2 rating. As one can see for about 75% of the results each round *at least* 75% of the remaining edges are removed from the graph, which is far more than the expected 50%. More interestingly, during the first few rounds (2–3), the fraction of removed edges decreases slightly, but then starts to increase quite fast. This indicates that, in practice, the number of rounds is *less than logarithmic* in the number of edges. This observation is also the case for the most extreme outliers. Which in some cases only remove about 20% of the remaining edges during a single round, but then start to remove more and more edges during the following rounds.

In case of 4 rounds the *outliers* with less than 50% removed edges are coarsened versions (levels 5, 6 and 9) of the add20 Graph from Walshaw's collection. In case of 5 rounds the outliers result from the add20 graph (level 5 and 7) and from the memplus graph (level 6 and 7), also from Walshaw's collection. That is also the case for the remaining results. Whenever less than 50% of the edges are removed, those results come from coarsened versions of the add20 or memplus graph.

Figure 4.4.18 shows the results when using random edge weights. As one can see the results are pretty *similar* to the ones we have seen before. For the majority of the results *at least 75%* of the remaining edges are removed during each round. Again we see a slight decrease of removed edges during the first few rounds and then again an increase for the remaining rounds. Although this time we do not see any cases where less than 50% of the edges are removed (also the case for the missing results). The most extreme outlier has about 49% remaining edges.

For the local tree algorithm we only show one of the results in Figure 4.4.19. This figure shows the worst case that we have observed for the local tree algorithm. The local tree algorithm required at most 5 rounds, although for the majority of the graphs only 3



(a) 4 Rounds

(b) 5 Rounds

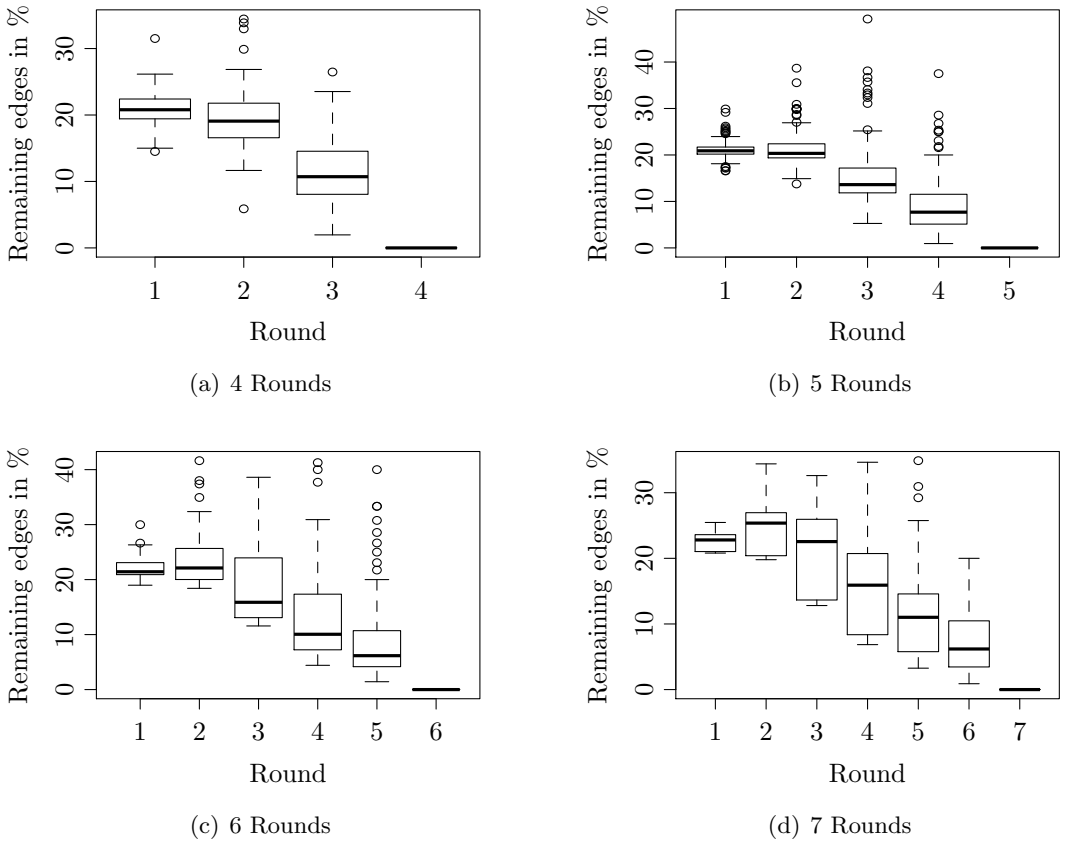
(c) 6 Rounds

(d) 7 Rounds

Figure 4.4.17.: Edge Development of the local max algorithm, with expansionstar2 as edge rating.

rounds. As one can see in Figure 4.4.19 usually *more than 95%* of the remaining edges are removed during a single round and again this number increases. Although in this example there are 3 cases where less than 85% of the edges are removed, this happened for coarsened versions of the finan512 graph from Walshaw's collection. In the missing cases at least 90% of the edges where removed during each round, and again for the majority of the results this number was larger than 95%.

Although the results indicate at most a *logarithmic number of rounds* in the number of edges, the number does *not solely* depend on the number of edges. It most likely also depends on the *structure* (e.g. average vertex degree) of the graph. This can be seen in Figure 4.4.20. We compare 4 different kinds of graphs in this figure: Random geometric graphs, Delaunay graphs, 2D grids and complete graphs.



(a) 4 Rounds

(b) 5 Rounds

(c) 6 Rounds

(d) 7 Rounds

Figure 4.4.18.: Edge Development of the local max algorithm, with random edge weights.

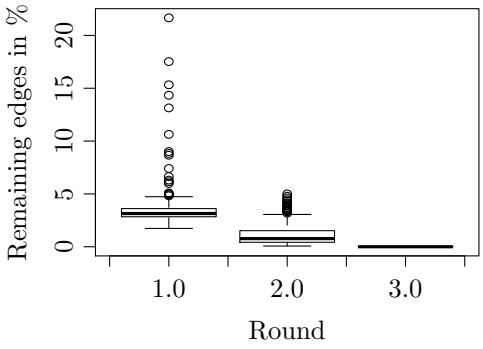


Figure 4.4.19.: Edge Development of the local tree algorithm, with expansionstar2 as edge rating.

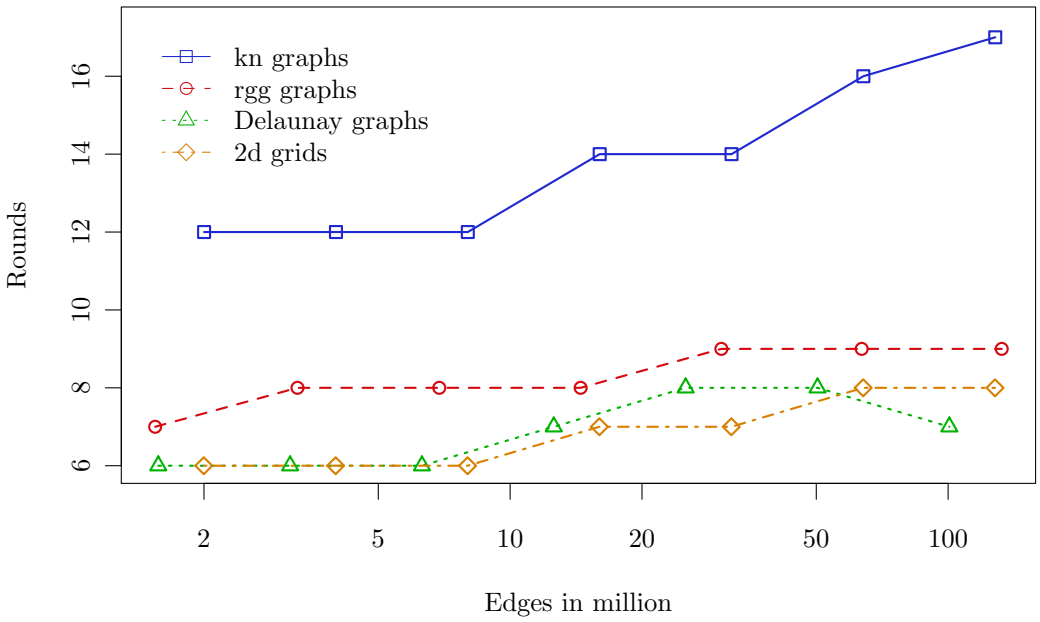


Figure 4.4.20.: Number of rounds of the local max algorithm for different kinds of graphs.

4.4.3. Time per Edge

We showed that the expected runtime for slight variations of the local max and local tree algorithm is linear in number of edges (Theorem 4.2, Theorem 4.3.2). The results from the previous section, that the number of rounds is logarithmic, also indicate that the runtime might be linear.

To verify the *linear runtime* we have a look at the time spent per edge. If we really have a linear runtime, then the time per edge should be constant. Because the number of rounds seems to not only depend on the number of edges, we have decided to look at the runtime of graphs of the same kind. The considered graph types are: Random geometric graphs, Delaunay graphs, complete graphs and 2D- and 5D-grids.

Figure 4.4.21 shows the results for random geometric graphs, for both the local max and the local tree algorithm and for random edge weights and the expansionstar2 rating. For both algorithms the runtime is about the *same* for both edge ratings. The local tree algorithm shows a *slight decrease* in the time spent per edge, whereas the local max algorithms shows a *slight increase* in the time spent per edge.

For Delaunay graphs the algorithms *do not show different runtimes* for different edge ratings (Figure 4.4.22). Except for the first number of edges, for both algorithms the *time spent per edge increases*. For the local max algorithm it only increases slightly. But for the local tree algorithm the time increases quite a lot for 10000 to 500000 edges, by a factor of about 2, and then the growth starts to slow down. It looks like this is caused by *cache misses*. For the Delaunay graph n11 (6127 edges) we have about 3.4

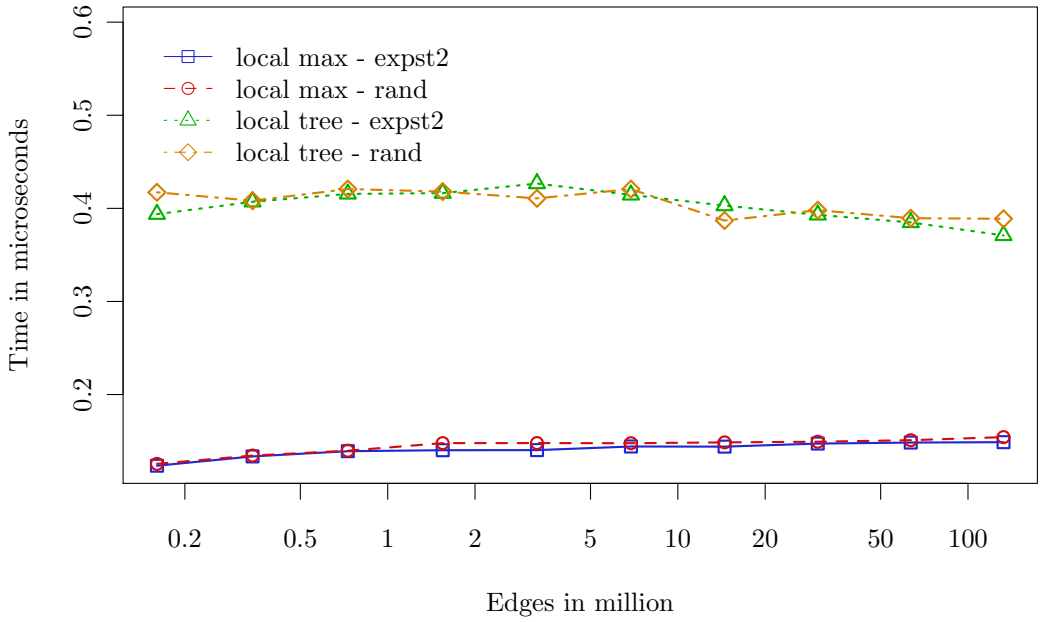


Figure 4.4.21.: Time spent per edge for random geometric graphs. Random edge weights and expansionstar2 rating.

L2 cache misses per edge and this increases up to 13.1 cache misses per edge for the Delaunay graph n18 (786 396 edges). The number of L2 cache misses per edge only increases slightly for larger Delaunay graphs.

Accessing the edges, except for tree edges, is done by a simple iteration through an array, thus this should not cause many cache misses and especially not an increase in the number of cache misses per edge. But for consecutive accesses to the candidate array and the matched vertex array it is likely that those accesses are not close to each other, e.g. the two end vertices of an edge might have a large difference in their vertex IDs or the IDs of end vertices of consecutive edges differ a lot. For a smaller number of vertices this is not that problematic because larger portions of those arrays fit into the cache and thus reducing the number of cache misses. A similar behaviour can be observed for the local max algorithm. This also suggests that the increase in cache misses is mainly caused by the access to the candidate and matched vertex array and not by the computation of matchings for the local trees.

Figure 4.4.23 shows the results for the local max algorithm for complete graphs and 2D- and 5D-grids with random edge weights as the input. For complete graphs we see a *really good result* with only a slight increase in the time spent per edge. The 5D-grid shows overall a rise by a factor of 1.15, for the 2D-grids we see *good* runtime behaviour except for the step from 32 million edges to 64 million edges. The number of L2 cache misses per edge increases in this case from 5.9 to 6.4.

The results for the same graphs as the input for the local tree algorithm are shown

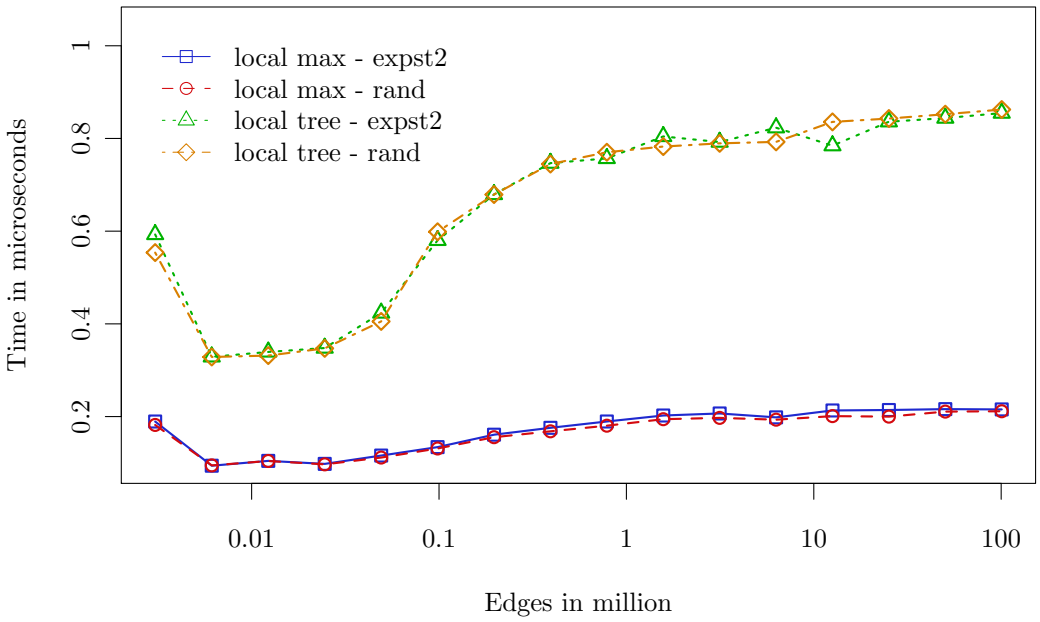


Figure 4.4.22.: Time spent per edge for Delaunay graphs. Random edge weights and expansionstar2 rating.

in Figure 4.4.24. The runtimes per edge for the grid graphs *increases slightly*. In total at most by a factor of 1.26. For the complete graphs we see a *rise* by a factor of 2.6, although the number of edges increases at the same time by a factor of 128. This rise is probably not cause by cache misses caused by accesses to the candidate array or matched vertices array. If this would be the cause we should see a similar behaviour for the local max algorithm.

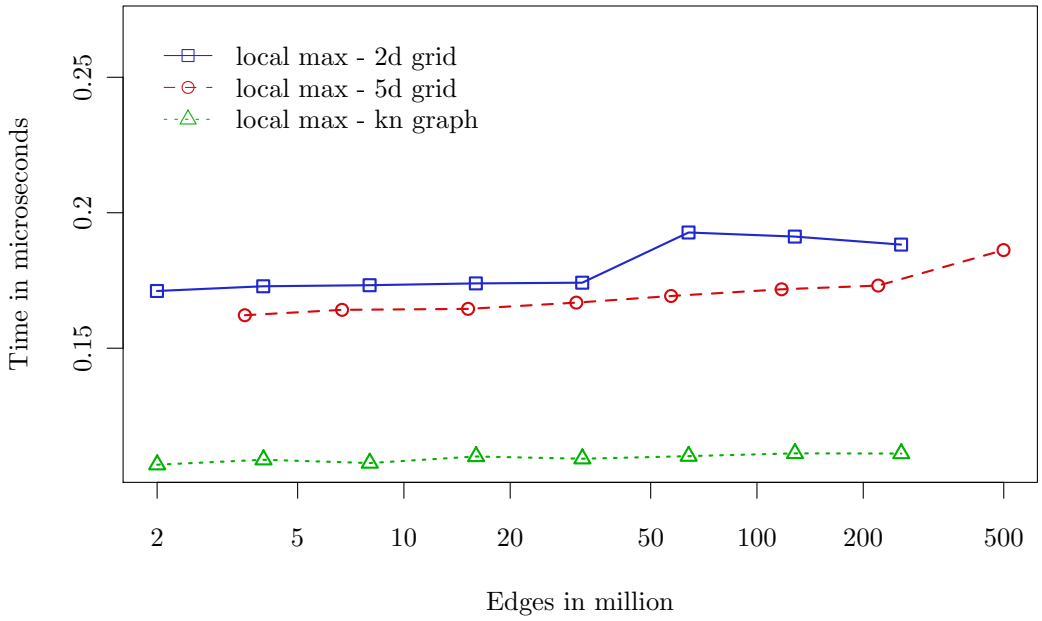


Figure 4.4.23.: Time local max spent per edge for 2D-, 5D-grids and complete graphs. Using random edge weights.

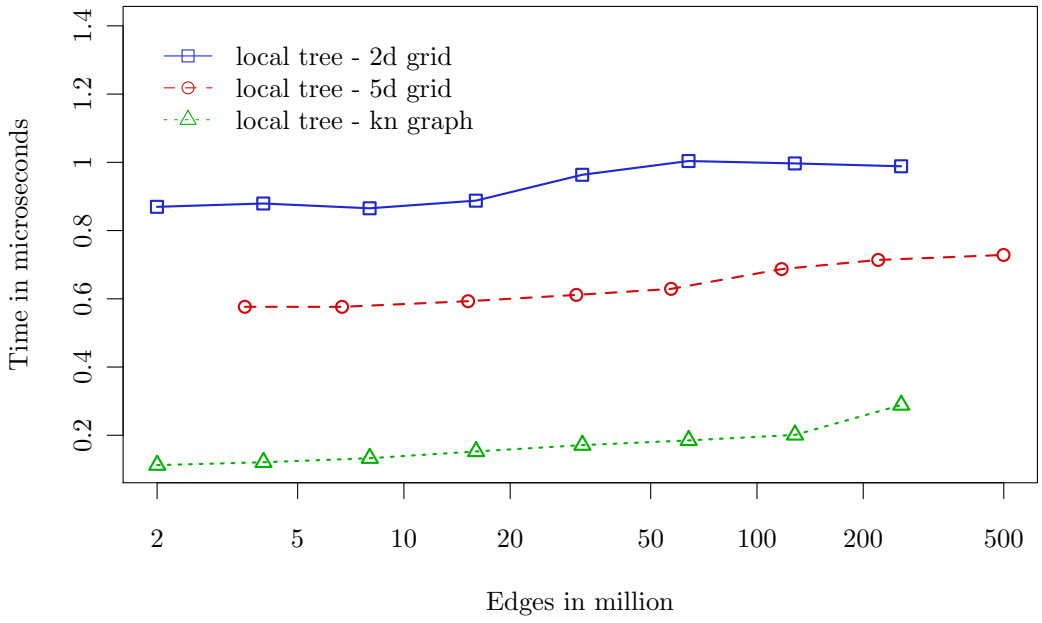


Figure 4.4.24.: Time local tree spent per edge for 2D-, 5D-grids and complete graphs. Using random edge weights.

4.4.4. Depth and Size of Local Trees

As we have mentioned before, we are going to describe parallel versions of the local max and local tree algorithms. Although in case of the local tree algorithm this might be problematic, since the described algorithm for computing maximum weighted matchings of trees is inherently sequential. But that is only really problematic if a single tree is located on several processes, in this case processes have to wait for other processes to finish their local computations.

To see if that is going to be a problem in practice, we observed the size and depth of local trees. The more interesting of those values is the depth of a tree. It gives an upper bound of processes which might have to wait for each other. Only along paths from leaves to the root vertex processes have to wait for each other (during the bottom up and top down phases). Computations along such a path cannot be done in parallel. The size of a tree gives an upper bound on the number of processes of a tree. So it is more a measurement for the required communication. Computations on independent subtrees can be done in parallel.

In this case we only considered the graphs from the 10th DIMACS Implementation Challenge and their coarsened versions using random edge weights and the expansion-star2 rating.

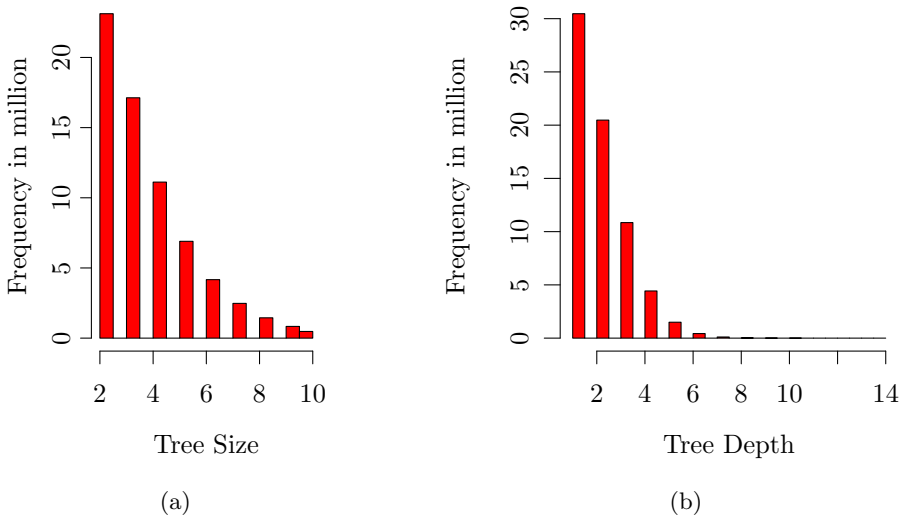


Figure 4.4.25.: Sizes and depths of local trees. Using random edge weights.

Figure 4.4.26 shows the results for random edge weights. In case of tree sizes the figure only shows the frequencies of trees with a size of 10 or less. The missing sizes only represent about 0.89% of all the trees. The average size of a tree is 3.7, the median is 3 and the upper quartile is 4. The five maximal observed tree sizes are $393, 409, 492, 549$ and 604. Those trees appeared in coarsened versions of the memplus graph.

The maximal observed tree depth is 14, but trees with a depth of 10 or larger only

account for about 0.0014% of all trees. The average tree depth is 1.94, the median is 2 and the upper quartile is 3.

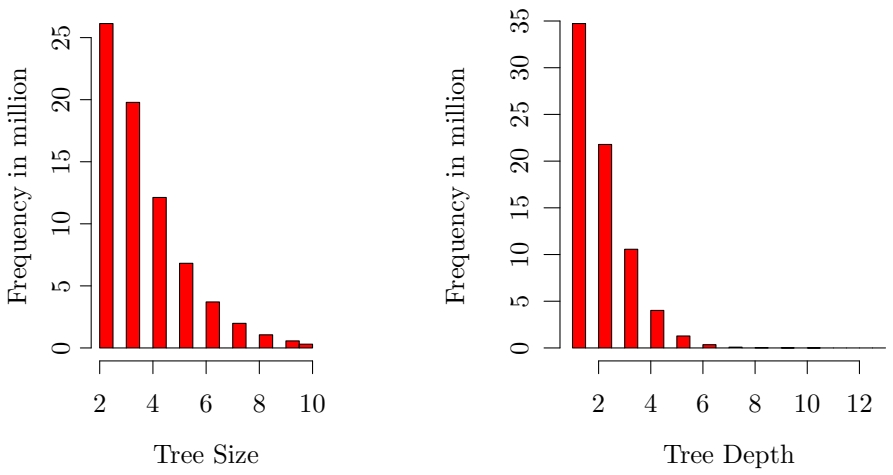


Figure 4.4.26.: Sizes and depths of local trees. Using expansionstar2 rating.

The results for the expansionstar2 rating are similar to what we have seen before (Figure 4.4.26). Again, Figure 4.4.26 does not show all results for the tree sizes. About 0.49% of the results of the tree sizes are missing.The maximum observed depth is 13, although the average depth is 1.87 and the median and upper quartile are both 2. Trees with a depth of at least 10 account only for about 0.0009% of all observed tree depths. For the sizes of the trees we have an average of 3.45 and a median and upper quartile of 3 and 4, respectively. That is similar to what we have seen for random edge weights, but the maximal tree sizes got bigger. The five largest tree sizes are 1533, 1752, 1771, 1880 and 2000, all observed for coarsened versions of the memplus graph.

Those results show that by far most local trees are fairly *small* and have *really small depth*, in fact there is no tree with a depth larger than 14. This indicates that in practice a *parallel version could work well* (Section 5.2).

5. Parallel Algorithms

In this section we discuss possible parallel versions of the local max and local tree algorithms from Chapter 4 to compute approximate maximum weighted matchings. The implementations are based on MPI (Message Passing Interface) [1].

Each process of the parallel environment gets a subgraph of the whole graph as its input, such that the subgraphs combined correspond to the initial graph. In our case we decided to use subgraphs that are defined by contiguous blocks of vertices, the sizes of those blocks differ at most by one. Each subgraph consists of the edges incident to the vertices of those blocks. Obviously each process might have edges that connect it with another process and the number of edges assigned to the different processes might differ significantly. The advantage of this approach is its simplicity.

Another approach would be to use a graph partitioner to compute those subgraphs, such that the size of the subgraphs is about the same on each process and the number of edges between different processes is minimized. Depending on the application this method might be reasonable, as long as the graph partitioner is fast enough. But if the parallel matching algorithm is supposed to be part of a graph partitioner (e.g. a coarsening step) then it is problematic to use a partitioner at first. We have introduced such an application in the introduction (Chapter 1).

5.1. Parallel Local Max Algorithm

As we have mentioned before Algorithm 4.1.1 can be easily parallelized using a BSP-style approach. The basic idea of the parallel Algorithms is shown in Algorithm 5.1.1.

Like in the sequential case at first we compute the set of locally heaviest edges. In the case of local edges ($local(E)$) it is easy to decide whether they are locally heaviest edges or not. Each process knows all the incident edges of its local vertices. More problematic are the cross edges ($cross(E)$). From now on we assume that the *first mentioned end vertex* of a cross edge is the local vertex and the other one the ghost vertex. To decide whether a cross edge $e = \{u, v\}$ is a locally heaviest edge we have to consider all the incident edges of u and v. But the process $p(u)$ of the local vertex u most likely does not know all the incident edges of the ghost vertex v. The process $p(u)$ can only decide if e is a *candidate* for a locally heaviest edge. That is the case when e is the heaviest incident edge of u.

For each such candidate $e = \{u, v\}$ we send the message $\langle req(e) \rangle$ to the process $p(v)$ of v, to tell $p(v)$ that e is a candidate on process $p(u)$ (line 6). After request messages have been sent for each candidate, the processes receive all request messages that were sent to them during the current round. For each incoming message $\langle req(e = \{v, u\}) \rangle$ the

Algorithm 5.1.1 Compute an approximate weighted matching in parallel

parallel_local_max$(G_p = (V, E))$:

```
1:  M = ∅
2:  while E ≠ ∅ do
3:    L = {e = {u, v} ∈ local(E) | e maximal at u and v}
4:
5:    C = {e = {u, v} ∈ cross(E) | e maximal at u and v ghost vertex}
6:    for all e = {u, v} ∈ C do
7:      send message ⟨req(e)⟩ to process of ghost vertex v
8:    while incoming message ⟨req(e = {v, u})⟩ of current round exists do
9:      if e maximal at v then
10:       L = L ∪ {e}
11:
12:   M = M ∪ L
13:
14:   N = set of newly matched vertices adjacent to a ghost vertex
15:   for all u ∈ N do
16:     send message ⟨matched(u)⟩ to each process of adjacent ghost vertices of u
17:   while incoming message ⟨matched(u)⟩ of current round exists do
18:     set u to matched
19:
20:   remove_edges_incident_to_matched_vertices(G_p)
21: return M
```

algorithm checks if e is the heaviest incident edge of vertex v, if that is the case then e is added to the set of locally heaviest edges. Obviously in this case another request message was sent from process $p(v)$ to process $p(u)$. Figure 5.1.1 shows an example where a cross edge is a candidate of one process but not the other one.

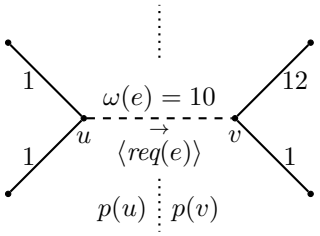


Figure 5.1.1.: Communication example for a candidate edge.

In this example one message is sent from process $p(u)$ to process $p(v)$, because the cross edge e is the heaviest incident edge of vertex u. But no message is sent from $p(v)$ to $p(u)$, because e is not the heaviest incident edge of v.

After all locally heaviest edges have been identified they are added to the set of matched edges. At this stage the sequential algorithm removes all edges incident to

matched vertices from the graph. However in case of the parallel version that is not possible, because a process might not know that one of its ghost vertices is matched on its originating process. Consider the example from Figure 5.1.1 again. In this case the vertex v is matched on process $p(v)$, but the incident cross edge e is not matched. Therefore e would be removed from $p(v)$'s subgraph but not from $p(u)$'s subgraph. This would cause the algorithm not to terminate on $p(u)$, because the edge e will always be considered as a candidate.

Before removing incident edges of matched vertices we send for each matched vertex u, that is incident to a ghost vertex, the message $\langle matched(u) \rangle$ to each process of its adjacent ghost vertices (line 15). Afterwards each process receives all messages $\langle matched(u) \rangle$ that where sent to it and marks the vertices u as matched. Now each process is able to safely remove edges incident to matched vertices.

Each round of the algorithm consists of two different BSP phases. At first locally heaviest edges and candidate edges are computed and subsequently a communication phase starts to decide whether a candidate edge is a locally heaviest edge. During the second BSP phase we add the locally heaviest edges to the matching, then compute all matched vertices incident to ghost vertices and send messages to the corresponding partners. The difference to a real BSP program is that we do not enforce a step where each process has to wait until all communication of the current phase has finished.

5.1.1. Implementation Details

We only talk in detail about the used graph data structure and the implementation of methods involving communications. The implementations of the other methods are similar to implementations that we have seen for the sequential algorithm (Algorithm 4.1.2).

Like in the sequential case, each subgraph is represented by a simple array storing all edges of the graph. However we use *two arrays*, one is used to store *local edges* and the other one used to store the *cross edges*. The arrays are divided into two parts. The first part is used to store the active edges and the second part is used to store inactive (deleted) edges. Deactivating an edge is done in the same way as it is done by the sequential algorithm. In the case of cross edges we ensure during the construction of the graph that the first vertex of the cross edge is always the local vertex and the second vertex is the ghost vertex. This simplifies the decision which of the vertices is the ghost vertex. Additionally we store information about the vertices. Instead of using the vertex IDs provided by the input graph (called *global IDs*) each process maps those IDs to the range $[0, n_p)$, where n_p is the number of distinct vertices of the subgraph. Each subgraph of a process is defined by a contiguous block $[a, b)$ of vertex IDs. Each global vertex ID $i \in [a, b)$ is mapped to the local vertex ID $i - a$. And the IDs of the ghost vertices are mapped to the remaining $n_p - (b - a)$ local IDs in the order in which they appeared during the construction of the graph structure. The advantage of the mapping approach is that we are now able to use simple arrays to store information about matched vertices and the maximal incident edges, like we do in the sequential algorithm. Using the array approach with global IDs is not appropriate because the range of the IDs of ghost vertices might be a lot larger than the number of distinct vertices of the subgraph

of a process, therefore using much more memory than necessary. However we still need (as we see later) a possibility to transform local IDs to global IDs and vice versa. The *transformation* between the local IDs of *local vertices* and the corresponding global IDs is done by either adding a to the local ID or subtracting a from the global ID. Also the transformation of a local ID of a ghost vertex to the global ID is easy, one can just use an array in the size of the number of ghost vertices. More problematic is the other direction for ghost vertices. Because of the large possible range of global IDs of ghost vertices it is not a good idea to use an array. Instead we decided to use a hash table (boost::unordered_map) to transform global IDs of ghost vertices to their corresponding local IDs.

Additionally we store for each ghost vertex the ID of the corresponding process, using an array for $O(1)$ look ups. We also need a way to know if we still require to communicate with a *partner process*, i.e. there is at least one cross edge to this process. We call a partner *active* if there are active cross edges to this partner. An active partner indicates that there is possible communication with this partner. During the construction of the graph we count for each process the number of cross edges to this process and whenever a cross edge is delete we decrement this number by one. So if the number of cross edges to a process is not 0, we know that this partner is still active. This does not affect the constant runtime of deleting an edge.

Now we get to the actual implementation of Algorithm 5.1.1. Before the computation of the matching we initialize two arrays on each process to keep track of matched vertices and the heaviest incident edge of a vertex. We have already seen this in the sequential case. Each round of the computation of the matching consists of four steps (see Algorithm 5.1.2).

Algorithm 5.1.2 Compute an approximate weighted matching in parallel

parallel_local_max_implementation($G_p = (V, E)$):

1: $M = \emptyset$
2: $C(n, dummy)$ // *Initialize candidates*
3: $m(n, false)$ // *Initialize matched vertices*
4: **while** $E \neq \emptyset$ **do**
5: *set_candidates*(G_p, C)
6: *add_locally_heaviest_edges_to_matching*(G_p, m, C, M)
7: *exchange_information_about_matched_vertices*(G_p, m)
8: *remove_edges*(G_p, m, C)
9:
10: **return** M

At first we set the maximal incident edge (*candidate*) of each vertex. Setting the candidates of local vertices is really simple and it is done the same way as in the sequential case, but this time we chose to use vertex IDs to break possible ties. Tie breaking based on vertex IDs makes it harder to distinguish multi-edges, therefore we decided not to allow multi-graphs. For more information about tie breaking using vertex IDs see Appendix C.1.

As mentioned before to decide if a cross edge is a locally heaviest edge we have to know if it is the heaviest incident edge of both end vertices. This computation for the local vertices of the cross edges can be done independent from the ghost vertices. The computation of the candidate of ghost vertices is based on the observation that each ghost vertex u is a local vertex on the process $p(u)$. This process computed the locally heaviest incident edge $e = \{u, v\}$ of u. So if the edge e is a cross edge then process $p(u)$ can inform the process $p(v)$ that e is the heaviest incident edge of its ghost vertex u. This does not necessarily initialize all ghost vertices on all processes, e.g. the heaviest incident edge might not be a cross edge or a vertex u is a ghost vertex on more than one processes. But in those cases the heaviest incident edge is still set to the dummy edge and thus will not affect the result. The implementation can be seen in Algorithm 5.1.3.

Algorithm 5.1.3 Set candidate edges of ghost vertices

set_candidates_of_ghost_vertices(G_p, C):

```
 1: for all remaining cross edges e = {v, u} of G_p do
 2:    if e == C[v] then
 3:       messages_for_proc[proc_of(u)].add(⟨req(e)⟩)
 4:
 5: for all active partners p do
 6:    send messages_for_proc[p] of type request to p
 7:
 8: for all active partners p do
 9:    incoming_msgs = receive messages of type request from p
10:    for all ⟨req(e = {u, v})⟩ ∈ incoming_msgs do
11:       C[v] = e
```

At first we iterate over all remaining cross edges $e = \{v, u\}$ and check if e is the heaviest incident edge (candidate) of the local vertex v, if this is the case then we add the message $\langle req(e)\rangle$ to a list of messages which are sent to process $p(u)$. This allows us to send all messages from process $p(v)$ to process $p(u)$ within a single MPI-message and thus reducing the number of MPI-messages. Obviously we only add messages to a message list of an active partner. After we have finished adding messages to those lists we can send them to the corresponding processes by just iterating over all active partners. Each message is of type *request* to be able to distinguish them from other kinds of messages. We not only send a message to an active partner with a non empty list but also to those active partners with an empty list. This is necessary because the partner process cannot know if it is going to receive an empty message. In the example of Figure 5.1.2, the process $p(u)$ does not send any messages to $p(v)$, but $p(v)$ cannot know this from its own knowledge about the graph. Therefore we decided to allow empty message.

To be able to set the maximal incident edges of the ghost vertices, each process receives from each active partner a bundle of messages of type *request* and then iterates over the single messages and sets the maximal incident edge of the ghost vertex. For the receive

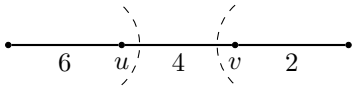


Figure 5.1.2.: Process $p(u)$ does not send a message to process $p(v)$, because edge $\{u, v\}$ is not the maximal incident edge of u.

operations we decided to use MPI's probe mechanism and a blocking receive operation. See Appendix C.2 why we decided to use this approach to receive messages.

As we mentioned before we distinguish between local and global vertex IDs. Because processes cannot know the local vertex IDs of ghost vertices on other processes we use the global vertex IDs for the communication. Before sending messages $\langle \mathit{req}(e = \{u, v\}) \rangle$ we transform the local IDs of u and v to their global IDs, to identify those vertices. When receiving the messages we transform the global IDs back to local IDs.

That was the first step of a single round. So far we have identified the heaviest incident edge of each vertex. This allows us to add the locally heaviest edges to the matching during the second step (*add_locally_heaviest_edges_to_matching*). Like in the sequential algorithm we iterate over all remaining edges (local and cross edges) and if an edge $e = \{v, u\}$ is the heaviest incident edge of both end vertices u and v we know that it is a locally heaviest edge and can add it to the matching. Additionally we mark those two vertices as matched.

Before we are able to remove edges incident to matched vertices we have to make sure that each process knows which of its *ghost vertices is matched* (Algorithm 5.1.4), that is done during the third step of a round. At first we iterate over all cross edges $e = \{v, u\}$ and check if the local vertex v is matched. If that is the case we add the message $\langle \mathit{matched}(v) \rangle$ to the list of messages for process $p(u)$. The vertex v is a ghost vertex of process $p(u)$. The computed messages are then sent in bundles to each partner, again we allow empty messages. It is possible that a bundle of messages contains a particular vertex more often than once (if there is more than one cross edge incident to this vertex), this does not affect the correctness but might increase the size of the messages. Using a preprocessing step we could remove duplicates. But sending those extra messages does not change the fact that not more than a constant amount of messages is sent for each cross edge during one round. After receiving the messages about matched ghost vertices we set each received vertex to be matched. Of course before sending the messages we have to transform the local vertex IDs to global IDs and IDs of received vertices are transformed into local IDs.

During the last step of a round (*remove_edges*) we remove the edges incident to matched vertices. This is done using the same procedure as used in the sequential algorithm. Again for each edge that is not incident to a matched vertex we reset the candidate of its end vertices, to guarantee that during next round old candidates will not influence the computation of new candidates.

Now lets have a look at the work of the parallel algorithm on a *single process*. The only problematic operations which could change the linear work (in number of remaining edges on this process) of a single round are those involving communication, those are

Algorithm 5.1.4 Exchange information about matched vertices

exchange_information_about_matched_vertices(G_p, m):

```
 1: for all remaining cross edges e = {v, u} of G_p do
 2:    if m[v] then
 3:       messages_for_proc[proc_of(u)].add(⟨matched(v)⟩)
 4:
 5: for all active partners p do
 6:    send messages_for_proc[p] of type matched to p
 7:
 8: for all active partners p do
 9:    incoming_msgs = receive messages of type matched from p
10:    for all ⟨matched(v)⟩ ∈ incoming_msgs do
11:       m[v] = true
```

the additions to the sequential algorithms. In the two cases (Algorithm 5.1.3 and Algorithm 5.1.4) when communication is involved we iterate at first once over the remaining cross edges to add messages to the bundles. Adding a message to a bundle only requires a constant amount of work (transforming of local vertex ID into a global vertex ID is done in constant time). Assuming that the work required to send a message is linear in the size of the message we get that the work required to send the messages of one round is at most linear in the number of remaining cross edges. The size of all bundled messages together cannot exceed the number of remaining cross edges. Also the empty messages, that we might send, will not affect this because the number of active partners cannot be larger than the number of remaining cross edges (assuming that it requires constant work to send an empty message). This general evaluation can also be applied when receiving messages, the total size or volume of incoming bundles cannot be larger then the number of remaining cross edges. Also setting candidate edges of ghost vertices or their state to matched only requires constant work. But before those operations we have to transform the global ID of a ghost vertex to its local ID which is done using a hash table. Hence we cannot guarantee constant work for this operation but, depending on the used hash table implementation, we expect constant work. Therefore the total work of a single round on one process is at best expected linear in the number of remaining edges. This gives that the total work of all processes combined of a single round is expected to be linear in the number of all remaining edges.

Obviously that is not necessarily the case for the runtime of a single round. For the analysis of the runtime we would have to respect the time that is required to transmit a message and more importantly the runtime is mainly affected by the largest partition of the graph. Our implementation of the graph distribution does not guarantee evenly sized subgraphs, therefore some processes might have to do more work than others.

Another interesting size is the volume of all messages combined. During a single round we might send at most four messages per cross edge, two request messages and two matched messages. But in the current implementation we send at least one matched

message over all rounds for each cross edge, because at least one of the end vertices of each cross edge must be matched, otherwise the matching would not be maximal. Hence the volume of all messages of one round is at most four times the number of remaining edges of the complete graph. The observation from the experimental section (Section 4.4) of the sequential algorithm that in each round at least half of the remaining edges are removed, suggests that the volume of all messages combined is linear in the number of edges of the input graph.

5.2. Parallel Local Tree

This section introduces the parallel version of Algorithm 4.3.1 (local tree algorithm). At first we show the basic structure of this algorithm and then proceed to talk in more detail about the parallel computation of maximum weighted matchings of trees.

5.2.1. Parallel Local Tree Algorithm

For the implementation of the parallel local tree algorithm we chose to use the same graph data structure as we used for the parallel local max algorithm from section Section 5.1. We also use for most parts functions that have been introduced in this section, except for the computation of maximum weighted matchings of trees, the computation of them is described in Sections 5.2.2 - 5.2.4.

Algorithm 5.2.1 shows the structure of the parallel local tree algorithm. At the start of each round we have to compute the *heaviest incident edge* of each vertex. This computation is the same as the computation of candidates for Algorithm 5.1.2. Each of those candidates is the heaviest incident edge of the corresponding vertex and therefore an edge of the forest of the current round. Adding those edges is done by the function *get_tree_edges* which simply iterates over each remaining local edge and cross edge $e = \{u, v\}$ and checks if e is the candidate of u or v, if so the edge is either added to the *local_edges*-list or the *cross_edges*-list. Using those two lists we can use the function *get_matching_of_parallel_forest* (Algorithm 5.2.2), which is described in the next section, to compute the *maximum weighted matching of the forest* defined by those two lists. In the actual implementation we set the matched vertices during the computation of the forest algorithm. Finally before *removing edges* incident to matched vertices we exchange information about matched vertices like we did in Section 5.1 using Algorithm 5.1.4. This is still necessary, e.g. consider the example from Figure 5.2.3. There are two processes, the process p_0 has the local vertices r, s and t and process p_1 has the local vertices u, v and w. Obviously there are two local trees, one on each process. The process p_1 matches the edge $\{v, w\}$ and process p_0 matches the edge $\{s, t\}$. Hence the ghost vertex t of process p_1 is matched but without the exchange operation p_1 would not know about this matched vertex and p_1 would not remove the edge $\{t, u\}$ from its subgraph.

Algorithm 5.2.1 Compute an approximate weighted matching in parallel

parallel_local_tree_matching$(G_p = (V, E))$:

1: $M = \emptyset$
2: *candidate_of_vertex*$[n, dummy]$
3: *matched*$[n, false]$
4: **while** $E \neq \emptyset$ **do**
5: $local\_edges = \emptyset$, $cross\_edges = \emptyset$
6:
7: *set_candidates*$(G_p,$ *candidate_of_vertex*$)$
8: *get_tree_edges*$(G_p,$ *candidate_of_vertex*, *local_edges*, *cross_edges*$)$
9:
10: $M' =$ *get_matching_of_parallel_forest*$($*local_edges*, *cross_edges*$)$
11: *set_matched_vertices*$(M',$ *matched*$)$
12: $M = M \cup M'$
13:
14: *exchange_information_about_matched_vertices*$(G_p,$*matched*$)$
15: *remove_edges*$(G_p,$ *matched*,*candidate_of_vertex*$)$
16:
17: **return** M

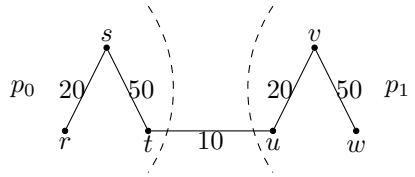


Figure 5.2.3.: Exchanging information about matched vertices is required by the parallel local tree algorithm.

5.2.2. Parallel Maximum Weighted Matching of a Forest

The computation of maximum weighted matchings of parallel forests is shown in Algorithm 5.2.2. At first we have to *compute the local trees* defined by the sets of local edges and cross edges (representing the tree edges). This time, unlike in the sequential case, we distinguish between *parallel trees*, those are the trees whose vertices are located on more than one processor, and *purely local trees*, those trees are only located on a single processor. The trees are represented by their root vertices, *parallel roots* and *local roots*, respectively. This computation is described in Section 5.2.3.

The *computation of maximum weighted matchings* of purely local trees is done as before and the computation of matchings of parallel trees is based on a parallel dynamic programming approach. See Section 5.2.4 for more details.

The parallelism of the computation of a maximum weighted matching of a forest comes from the fact that the matchings of the individual trees can be computed independent from each other. Further it is also possible to do computations on subtrees of a local

Algorithm 5.2.2 Compute the matching of a parallel forest

get_matching_of_parallel_forest(*local_edges*, *cross_edges*):

1: $local\_roots = \emptyset$
2: $parallel\_roots = \emptyset$
3: $forest = get\_parallel\_forest(local\_edges, cross\_edges, local\_roots, parallel\_roots)$
4:
5: $M = compute\_matching\_of\_parallel\_forest(forest, parallel\_roots)$
6: $M = M \cup compute\_matchings\_of\_local\_trees(forest, local\_roots)$
7: **return** M

tree independent from each other, if they are located on different branches of the local tree.

5.2.3. Computation of a Parallel Forest

Before we are able to compute a maximum weighted matching of a tree we need a data structure to represent this tree. Like in the sequential case we decided to use a simple *adjacency list* for the representation of a tree. However in the parallel case the creation of this graph is more complicated. We cannot simply choose an arbitrary vertex as the root of a tree, because the decision may depend on the decision of other processes. Therefore we have to choose a global root of the parallel tree, which defines the roots of the subtrees of each process. For example consider the tree from Figure 5.2.4.

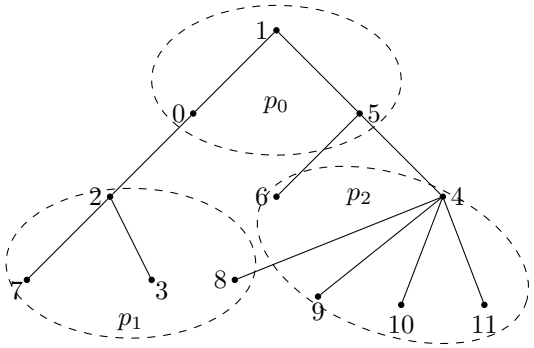


Figure 5.2.4.: Example for a parallel tree located on three processes.

This example shows a parallel tree which is located on three processes. On process 0 there is one subtree, two subtrees on process 1 and one or two subtrees on process 2 (depending whether we consider cross edges as part of the subtrees).

In this case vertex 1 is the root of the parallel tree, which defines the roots of the other subtrees. For the moment lets only consider local vertices as the roots of subtrees. Then we have on process 1 the vertices 2 and 8 as the roots of both subtrees and on process 2 the vertices 4 and 6 are the roots of the two subtrees.

If for example process 1 would choose the vertex 7 as the root of one of its subtrees

and process 0 the vertex 1 as a root, then we would get a damaged tree (a tree with two roots).

In the following we will describe a mechanism to decide on a global root of a parallel tree, and additionally it also gives us the roots of all the subtrees. This method also works for parallel forests.

Parallel trees consist of local edges and cross edges. The local edges and local vertices of a parallel tree define several subtrees, which are connected with each other by cross edges. We call these subtrees *border components*. Border components can be easily computed by computing the connected components defined by the local edges. A border component B_a is a *partner* of a border component B_b if they are connected by a cross edge. Figure 5.2.5 shows the border components for the tree from Figure 5.2.4. As one can see it is possible that a border component consists of a single local vertex. The border component B_1 is located on process 0, border components B_2 and B_3 are located on process 1 and the last two border components B_4 and B_5 are located on process 2. For more details on the structure and computation of border components see Appendix D.1.

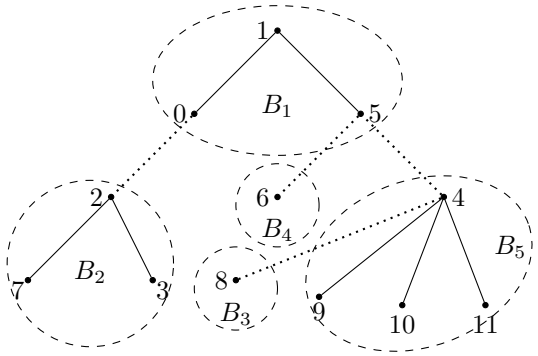


Figure 5.2.5.: Border components of Figure 5.2.4

The basic idea for the computation of a global root vertex of a parallel tree is that at first each border component chooses a local vertex which is a candidate for the root vertex. The root vertex will be the candidate with the *smallest ID*. After the selection of candidates each border component sends the ID of its candidate to all of its partners and receives messages from them. If one partner sends an ID smaller then the one of the candidate then the candidate is updated. In the case of an update each border component also stores from which partner it received the ID of the new candidate. The algorithm should stop when each border component received the smallest vertex ID. For more details on the termination of this process and how messages are bundled to reduce the number of MPI-messages see Appendix D.1.

After the computation of the global root it is easy for each process to choose the correct root vertex for each subtree represented by a border component. If the subtree contains the global root, then the global root is the root of the subtree. All other subtrees choose the ghost vertex of the cross edge which connects the subtree with the partner from where it received the ID of the global root.

Algorithm 5.2.3 summarizes all the necessary steps to compute a parallel forest from the sets of local and cross edges. At first we compute the border components, then we use the components to compute the global root of the local trees and finally use the result of this process to build the parallel trees.

Algorithm 5.2.3 Compute a parallel forest from sets of edges

get_parallel_forest(*local_edges*, *cross_edges*, *local_roots*, *parallel_roots*):

1: $bcs = \emptyset$
2: *get_border_components*(*local_edges*, *cross_edges*, *local_roots*, *bcs*)
3: *decide_on_root*(*bcs*)
4: add *local_edges* and *cross_edges* to *forest*
5: *parallel_roots* = *build_parallel_trees*(*forest*, *bcs*)
6: **return** *forest*

Appendix D.1 goes into more detail of the several steps of Algorithm 5.2.3, it also describes how to obtain the roots of purely local trees during the computation of the border components.

A Simpler Method to Decide on Root Vertices

The previously described method to decide on a root vertex for parallel trees requires up to l communication phases, where l is length of longest path within the tree defined by the border components. But that is a lot more than necessary, at least in our case. As we know from Lemma 4.6 each local tree computed by the local tree algorithm has a unique edge $e = \{u, v\}$ such that e is the heaviest edge of the tree and additionally all paths originating from e to leafs of this tree have decreasing edge weights. We can now define that the end vertex of e with the higher ID is the root of such a parallel tree. And for each border component the heaviest edge must be a cross edge, unless the component contains e. This heaviest cross edge is then the root of the tree of the border component, because we know that the root must be in the direction of this cross edge (the heavier edges of the tree are located in this direction). There is only one case which forces us to have some communication. It is possible that the heaviest edge e of the parallel tree is a cross edge, in this case the end vertex of e with the larger ID is the root. To find out if a cross edge is a root, each border component computes the local heaviest edge e' and then sends for each cross edge f the message $\langle \textit{not max, } f \rangle$, if f is not the heaviest edge and $\langle \textit{max, } f \rangle$ if f is the heaviest edge. When a border component receives a message $\langle \textit{max, } f \rangle$ and f is the local heaviest edge e then it knows that e is the global heaviest edge. Algorithm 5.2.4 shows the outline of an algorithm based on this idea. The main advantage of this algorithm is that it only requires a single round of communication apart from the fact that it is much simpler.

The reason why we introduced the other procedure to decide on a root vertex is because it is the more general version, it does not depend on Lemma 4.6 and therefore might be suitable in other situations where the root of parallel tree must be computed.

Algorithm 5.2.4 Compute the root edge of parallel trees of a forest

decide_on_root_simple(*bcs*):

```
 1: for all border components bc ∈ bcs do
 2:    bc.root_edge = heaviest_edge_of(bc)
 3:
 4: for all border components bc ∈ bcs do
 5:    for all partner p ∈ bc do
 6:       if p = bc.root_edge then
 7:          send message ⟨max, p⟩ to p
 8:       else
 9:          send message ⟨not max, p⟩ to p
10:
11: for all incoming messages ⟨type, p⟩ do
12:    bc = border_component_of(p)
13:    if type = max and bc.root_edge=p then
14:       bc.is_global_max = true
```

Also we used the more complex version during our experiments, because we were not aware of the simpler algorithm at this time.

5.2.4. Parallel Dynamic Programming

Now that we computed all the trees of the forest we can start computing the maximum weighted matchings of the trees. The matchings of purely local trees can be computed using Algorithm 3.3.1 from Section 3.3. The computation of matchings of parallel trees is based on the following idea. Each parallel tree consists of several subtrees, which themselves define a tree, the *border component tree*. Consider the example from Figure 5.2.5, the border components are the vertices of the tree which are connected by the cross edges. Like the sequential algorithm, the parallel version also consists of a bottom up phase and a top down phase.

During the *bottom up phase* each subtree propagates the two possible optimal results (the incoming edge is matched or not matched) to its parent. The result of a subtree only depends on the results of its own subtrees. Therefore a subtree is able to compute its result as soon as it receives the results of all of its subtrees that are located on another process. So at first the leaves of the border component tree send their results to their parents. Because those leaf-trees do not depend on other subtrees. Then the next batch of subtrees, which no longer depend on other subtrees, compute their results and send them to their parents. And so on, until the root subtree is able to compute its result.

The next step is the *top down phase*. This phase is used to add the edges to the matching. During this phase the subtrees whose roots are not the global root of a parallel tree depend on information from their parent. Those trees have to know if the cross edge to their parent is matched. In Figure 5.2.5 the border component B_2 depends on the border component B_1. So the idea for the top down phase is for each border

component to wait for the result of their parent component and as soon as they received the information whether their incoming edge is matched, they start to add edges to the matching and send information to their child components.

Algorithm 5.2.2 summarizes the computation a maximum weighted matching of a parallel forest.

Algorithm 5.2.5 Compute the maximum weighted matching of a parallel forest

compute_matching_of_parallel_forest(*forest*, *parallel_roots*):

1: *fill_subtree_table_parallel*(*forest*, *parallel_roots*)
2: M = *get_matched_edges_of_parallel_forest*(*forest*, *parallel_roots*)
3: **return** M

For more information how the sending and receiving of messages, for both phases, in particular the bundling of messages into one MPI-message is handled see Appendix D.2.

5.3. Experimental Results

In this section we present the experimental results of our parallel algorithms. Most of the experiments where performed on the InstitutsCluster (IC1) of the Steinbuch Centre for Computing [14]. This systems consists of 200 computing nodes. Each node is made up by two Quad-Core Intel Xeon X5355 processors (2.667 GHz) and 16 GB of RAM. The nodes are connected by an InfiniBand 4x DDR network. Suse Linux Enterprise (SLES) 11 SP 1 is used as the operating system on each node. The programs were compiled using Open MPI 1.5.5 and the Intel C++ Compiler 12.1.3 with optimization -O3. The other system that we used for the experiments is the KIT-Hochleistungsrechner HP XC3000 (hc3) of the Steinbuch Centre for Computing [15]. This system's nodes are made up by two Quad-Core Intel Xeon E5540 processors (2.53 GHz) and 24 GB of RAM. The hc3 uses HP XC Linux for "High Performance Computing" as its operating system. The programs where compiled using the same compiler, MPI-version and compile options from the IC1.

We used up to 1024 processes for our experiments, where each process is assigned to a single core. Therefore the number of processes also corresponds to the number of used cores.

For our parallel experiments we again used graphs from the 10th DIMACS Implementation Challenge [5], but this time we only considered graphs with at least 20 million edges. We used random geometrics graphs (rgg_n_2_22, rgg_n_2_23, rgg_n_2_24), Delaunay graphs (delaunay_n23, delaunay_n24, delaunay_n25), street graphs (europe.osm, road_us) and web graphs (uk-2002, uk-2007). Additionally we also used 2D-grids, 5D-grids and complete graphs.

All experiments on graphs with less than 400 million edges where performed on IC1. This includes all random geometric graphs, Delaunay graphs, the uk-2002 graph and several complete and grid graphs. Experiments on those graphs were repeated 10 times and we used up to 512 processes, although usually not more than 256.

The hc3 was only used for the largest graphs. Experiments on these graphs were performed using up to 1024 processes. However we performed these experiments only once. For all presented experiments we mention explicitly if they were performed on the hc3, all the other presented experiments were performed on the IC1.

5.3.1. Weak Scaling

At first we show the weak scaling results of the parallel algorithms. From the DIMACS-graphs we only considered the random geometric and Delaunay graphs. The other graphs are not really suitable because there are either too few of the same kind or the sizes differ in a way that is not really suitable for weak scaling.

Parallel Local Max Algorithm

Figure 5.3.6 and Figure 5.3.7 show the results of parallel local max algorithm (Algorithm 5.1.1) for 2D-grids. In the case of perfect weak scaling we would expect that the *runtime remains constant* when increasing both the number of processes and the size of the graph. As one can see that is not the case in the left picture of Figure 5.3.6, at least not from the jump from 8 to 16 processes. In this case the *runtime increases* by a factor of 1.87. On the other hand the right picture shows for larger numbers of processes an *optimal result*. The jump in the left picture was probably caused by the difference in the *number of processes* which were assigned to a single node. For 8 processes we used two nodes of the IC1 and hence only four cores of the 8 available cores were used on each node. In case of 16 processes we again used two nodes and therefore all available 16 cores were used. Throughout the experiments when all cores of the nodes were used we usually saw a worse scaling than for smaller numbers of processes on the IC1. Especially for the grid graphs and random geometric graphs the drop of the scaling factor is quite huge. So it is quite likely that the memory bandwidth, on the IC1, became a bottleneck when all cores of a node were used. For smaller numbers of processes there is not a perfect weak scaling for 2D-grids, but it is not too bad. The runtime for the step from 1 to 8 processes increases only by a factor of 1.49.

Figure 5.3.7 shows results for larger 2D-grids. In those cases we see *almost optimal* weak scaling for 16 up to 1024 processes.

The results for the 5D-grids (Figure 5.3.8) are really similar to those of the smaller 2D-grids. In both cases the grids have similar numbers of edges. We see a big *rise* in the runtime from 8 to 16 processes (factor 2.39) but we see *good* weak scaling results for larger numbers of processes (right picture). Again in the case of 8 processes only 4 cores of a single node were used whereas for 16 processes all 8 cores of a single node were used. For a smaller number of processes the weak scaling is not as good as the one for 2D-grids but it is not as bad as the step from 8 to 16 processes. The runtime for the step from 1 to 8 processes increases by a factor of 1.88.

For complete graphs (Figure 5.3.9) the results are *not that good*. Especially for smaller numbers of processes. One factor for the bad scaling is probably the *huge number of cross edges* for complete graphs. For 2, 4, 8, 16 and 32 processes we have that about

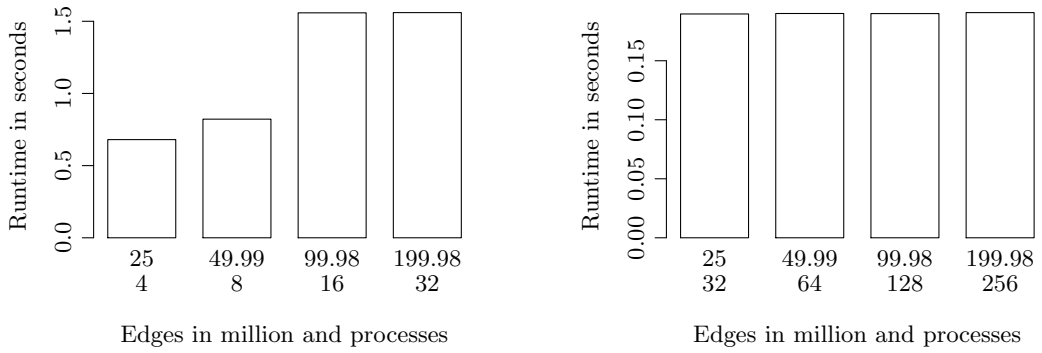


Figure 5.3.6.: Weak scaling results of the parallel local max algorithm on 2D-grids with random edge weights.

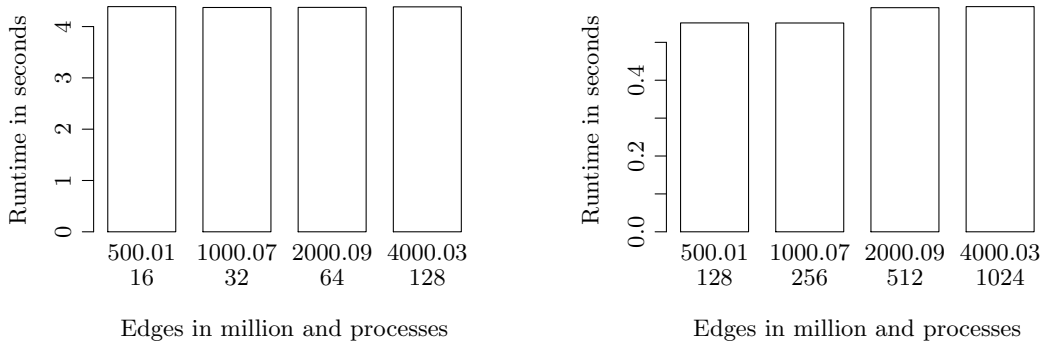


Figure 5.3.7.: Weak scaling results of the parallel local max algorithm on 2D-grids with random edge weights. (performed on hc3)

50%, 75%, 87.5%, 94% and 97% all edges are cross edges. The jump in the left picture from 4 to 8 processes is probably also caused by the fact that in case of 8 processes all 8 cores of one node were used. The right pictures shows good weak scaling behaviour for 128 to 512 processes but the there is a jump by a factor of 1.45 from 512 to 1024 processes. In both cases all cores of the used nodes were used, therefore the memory bandwidth cannot be an explanation for this jump. So maybe the bandwidth of the network became a bottle neck (we also see this jump when using the parallel local tree algorithm). Complete graphs need a lot more communication than grid graphs, because there are a lot more cross edges and there is communication involved between each pair of processes.

The weak scaling results for random geometric graphs are *not optimal*, but not too bad. The biggest jump is observed for the step from 4 to 8 processes, the runtime increases by a factor of 1.69. This is again the case where all cores of the node were

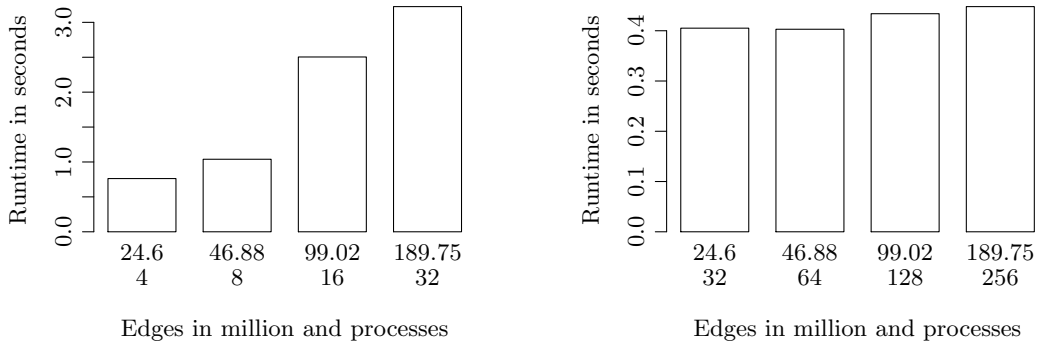


Figure 5.3.8.: Weak scaling results of the parallel local max algorithm on 5D-grids with random edge weights.

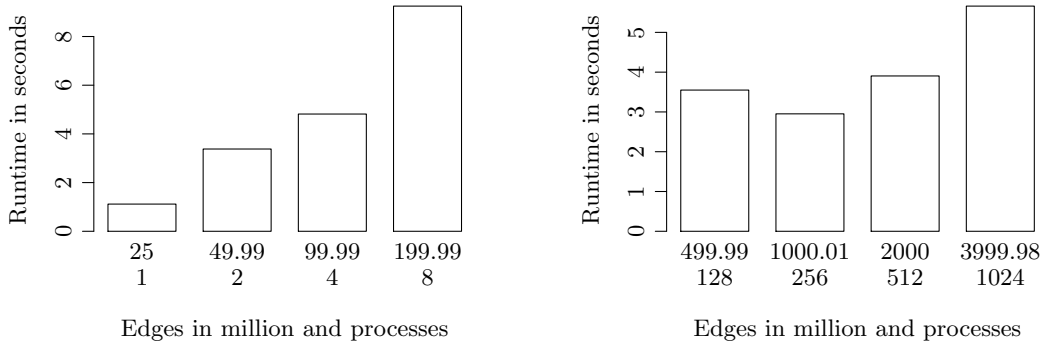


Figure 5.3.9.: Weak scaling results of the parallel local max algorithm on complete graphs with random edge weights. (right experiment was performed on hc3)

used. From one to 4 processes it increases by factor of 1.41. For larger numbers of processes the results are better. The runtime increases by a factor of 1.32 from 64 to 256 processes.

For Delaunay graphs we see pretty *bad weak scaling* especially for smaller number of processes. Similar to the complete graphs there are *a lot of cross edges* using our approach to partition the graphs. For 2, 4, 8 and 16 processes we have that about 25%, 37%, 39% and 40% of all edges are cross edges. In case of the random geometric graphs we only have that 0.03%, 0.06%, 0.1% and 0.2% are cross edges. Not using the trivial partitioning we can reduce the number of cross edges significantly. In case of the delaunay_n15 graph and using KaFFPa [31] we get that only about 0.3%, 0.7%, 1.1% and 1.9% are cross edges for 2, 4, 8 and 16 processes, respectively. So it is possible that a better partitioning results in a better scaling behaviour for Delaunay graphs.

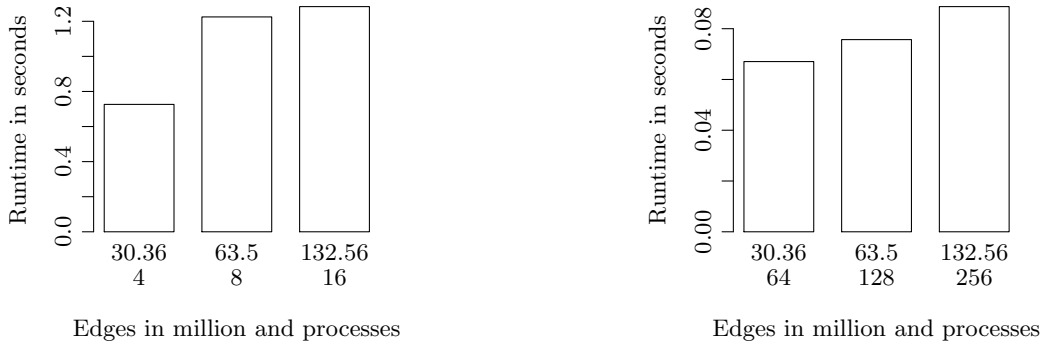


Figure 5.3.10.: Weak scaling results of the parallel local max algorithm on random geometric graphs (rgg_n_2_22, rgg_n_2_23 and rgg_n_2_24) with random edge weights.

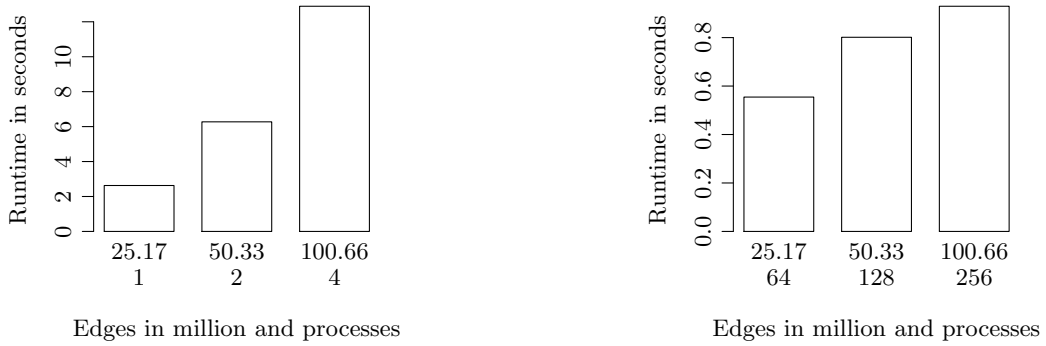


Figure 5.3.11.: Weak scaling results of the parallel local max algorithm on Delaunay graphs (delaunay_n23, delaunay_n24 and delaunay_n25) with random edge weights.

Parallel Local Tree Algorithm

The weak scaling results (Figure 5.3.12 – Figure 5.3.17) of the parallel local tree algorithm on the same graph-instances *do not show a significant different behaviour* than the results of the parallel local max algorithm. We see fairly *good weak scaling* for grid-graphs and random geometric graphs. For these graphs large runtime jumps are observed whenever all cores of a single node were used.

For the Delaunay and complete graphs we again see pretty *bad scaling*, especially for smaller numbers of processes.

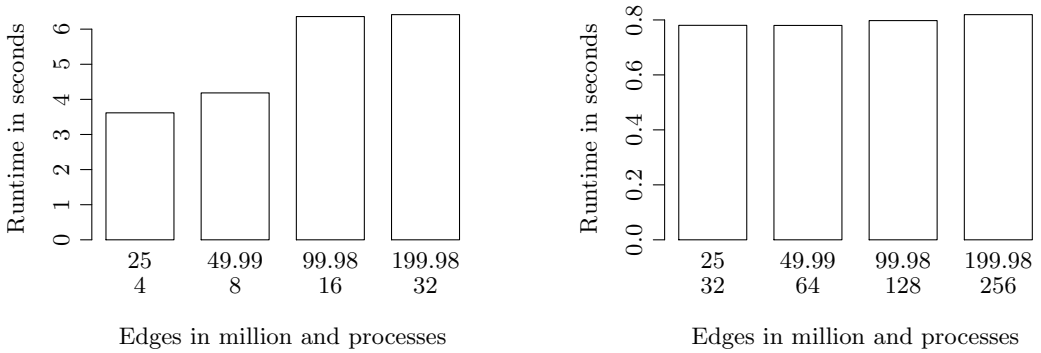


Figure 5.3.12.: Weak scaling results of the parallel local tree algorithm on 2D-grids with random edge weights.

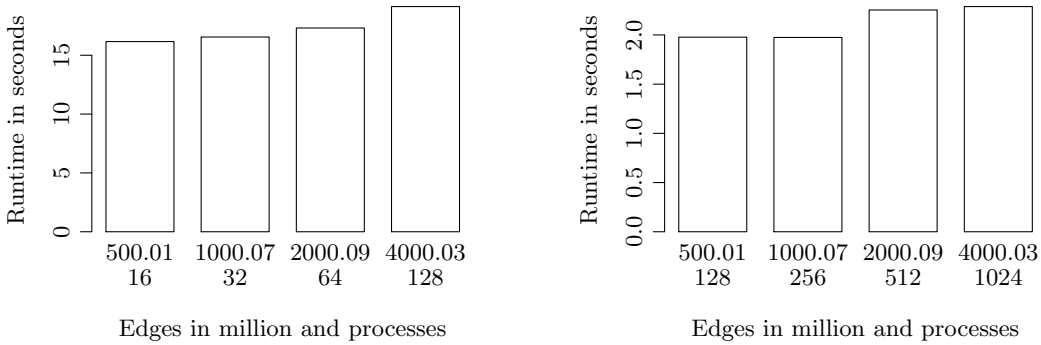


Figure 5.3.13.: Weak scaling results of the parallel local tree algorithm on 2D-grids with random edge weights. (performed on hc3)

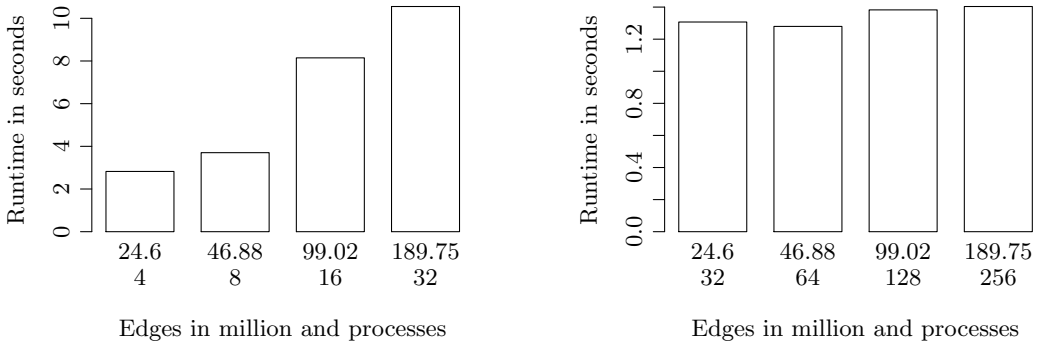


Figure 5.3.14.: Weak scaling results of the parallel local tree algorithm on 5D-grids with random edge weights.

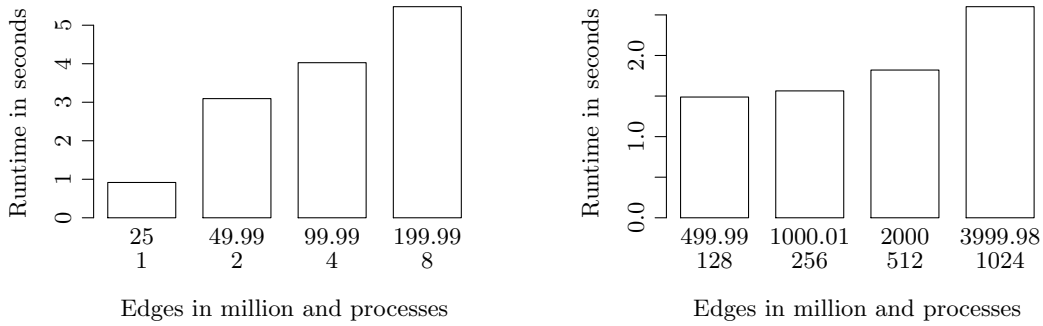


Figure 5.3.15.: Weak scaling results of the parallel local tree algorithm on complete graphs with random edge weights. (right experiment performed on hc3)

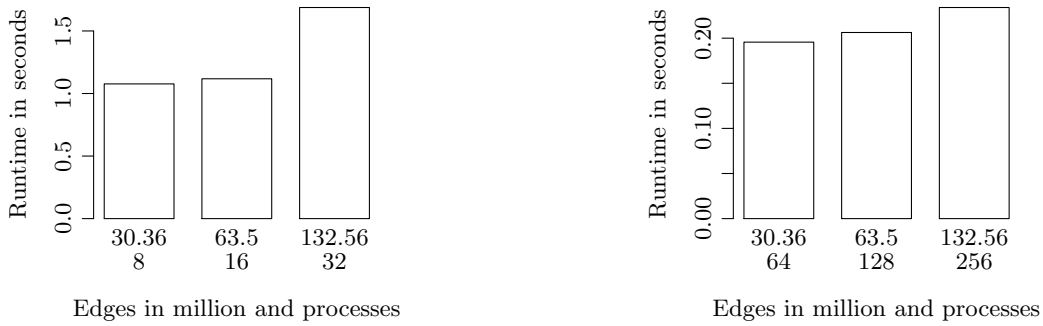


Figure 5.3.16.: Weak scaling results of the parallel local tree algorithm on random geometric graphs (rgg_n_2_22, rgg_n_2_23 and rgg_n_2_24) with random edge weights.

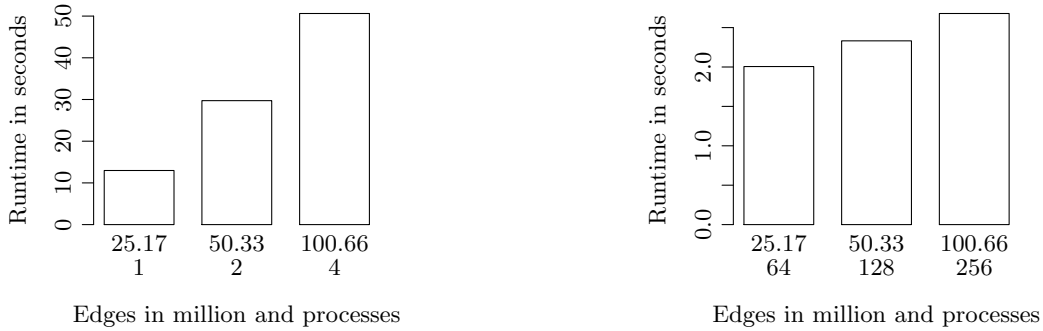


Figure 5.3.17.: Weak scaling results of the parallel local tree algorithm on Delaunay graphs (delaunay_n23, delaunay_n24 and delaunay_n25) with random edge weights.

5.3.2. Strong Scaling

We present strong scaling results for the two parallel algorithms in this section. In case of strong scaling one would hope to decrease the runtime of a parallel algorithm, on one particular graph, by the same factor by which we increase the number of processors. In our case we always double the number of processors. For good strong scaling the quotient between the runtimes using $p/2$ and p processors should be 2.

At first we present the strong scaling results for the parallel local max algorithm and afterwards the results for the parallel local tree algorithm.

Parallel Local Max Algorithm

Figure 5.3.18 shows the results for 2D-grids with about 50 million and 200 million edges. For 2 up to 8 processes the scaling is *not optimal* but it is still quite good. There is almost no speedup for the step from 8 to 16 processes. However starting with 16 processes all cores of the computing nodes were used, so the bad scaling for this step is probably caused by the memory bandwidth. For 32 to 256 processes we see *perfect scaling*.

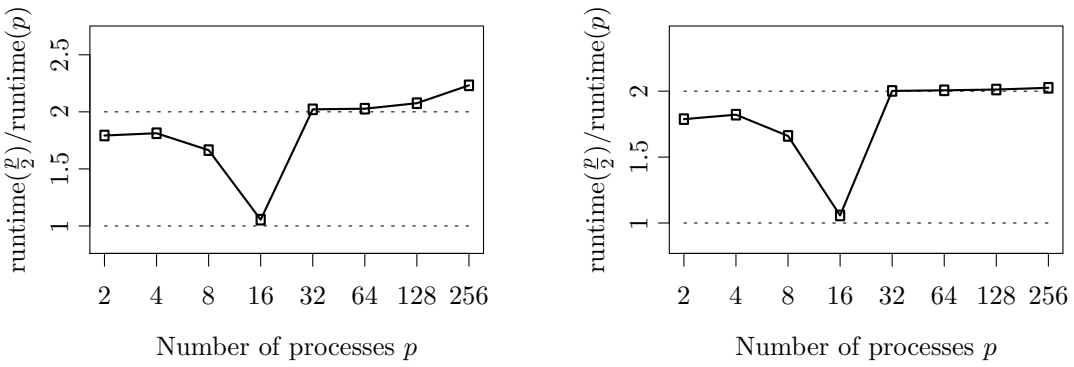


Figure 5.3.18.: Scaling results of the parallel local max algorithm on 2D-grids with random edge weights. Left: 2D-grid with 49 990 000 edges. Right: 2D-grid with 199 980 000 edges.

Figure 5.3.19 shows again results for 2D-grids, but this time for larger grids. For a 2D-grid with about 0.5 billion edges we *good scaling* for 32 up to 512 processes and no speed up from 512 to 1024 processes. We also performed experiments on 2D-grids with about 1 billion and 2 billion edges. In those two cases we observed the same behaviour. However as one can see in the right picture of Figure 5.3.19 we constantly see *good* scaling for a 2D-grid with about 4 billion edges. Hence the other graphs might have been too small for this amount of processes (Gustafson's law).

The strong scaling results for 5D-grids (Figure 5.3.20) are similar to the results from the smaller 2D-grids. For small numbers of processes (up to 8) the algorithms scales *reasonably well*. For those numbers not all available cores of the nodes were used. Starting with 16 processes all cores of the nodes were used and again we do not see any

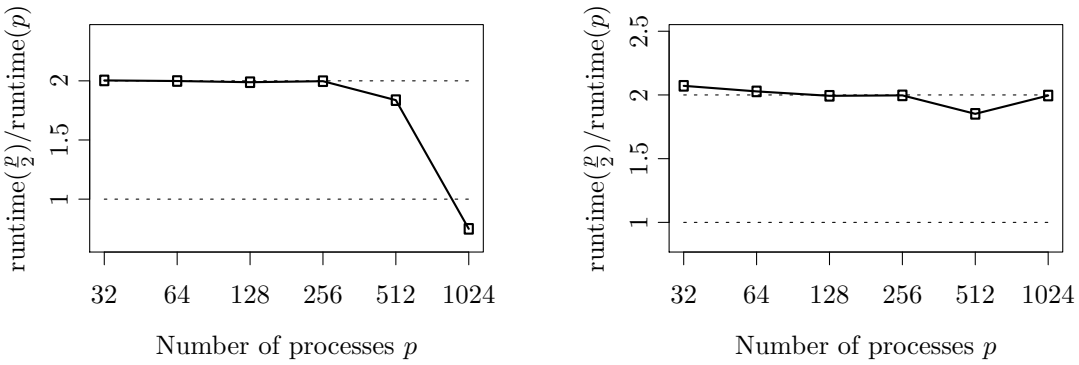


Figure 5.3.19.: Scaling results of the parallel local max algorithm on 2D-grids with random edge weights. Left: 2D-grid with 500 007 064 edges. Right: 2D-grid with 4 000 025 124 edges. (performed on hc3)

speedup for this step. For larger number of processes (up to 256) the algorithm *scales well* especially for the bigger graph.

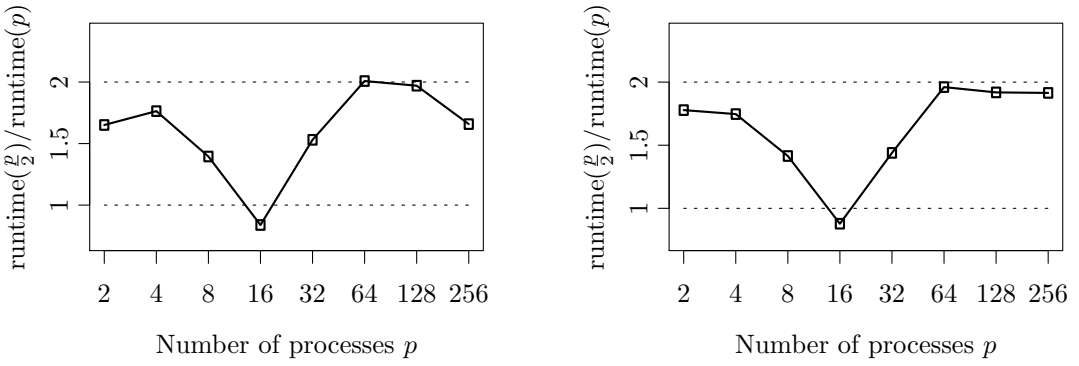


Figure 5.3.20.: Scaling results of the parallel local max algorithm on 5D-grids with random edge weights. Left: 5D-grid with 46 875 000 edges. Right: 5D-grid with 189 747 360 edges.

For a complete graph with about 12.5 million edges (Figure 5.3.21) the parallel local max algorithm scales *pretty bad*, actually in many cases we see a longer runtime. For a larger complete graph with about 200 million edges the scaling behavior is a lot better but still the scaling is *not at all optimal*. We only see a good scaling behaviour for 16 to 64 processes. The behavior for smaller numbers of processes is probably because here we see a large increase of cross edges. In case of 16 processes about 94% of all edges are cross edges, so this number cannot increase a lot more for more processes. Also the drop at 8 processes is probably because from there on all cores of the nodes were used.

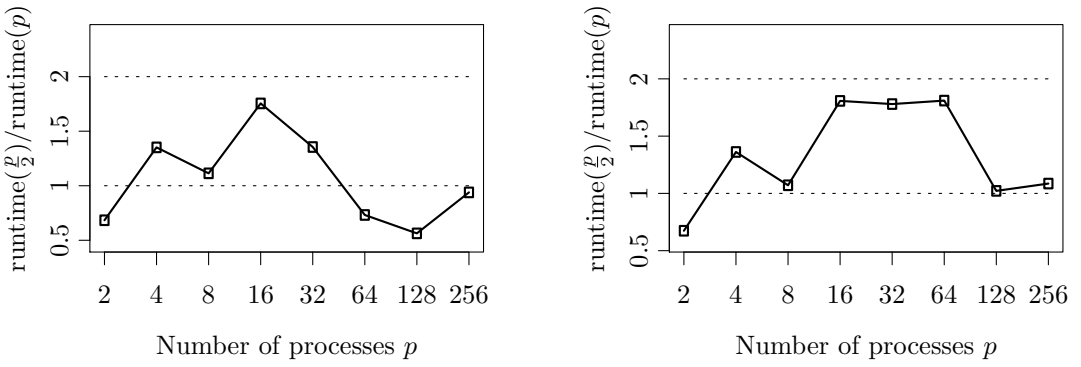


Figure 5.3.21.: Scaling results of the parallel local max algorithm on complete graphs with random edge weights. Left: complete graph with 12 497 500 edges. Right: complete graph with 199 990 000 edges.

For a complete graph with about 500 million edges (Figure 5.3.22) the algorithms *scales well* for up to 256 processes and in the case of a complete graph with about 4 billion edges we see *good scaling* for up to 512 processes.

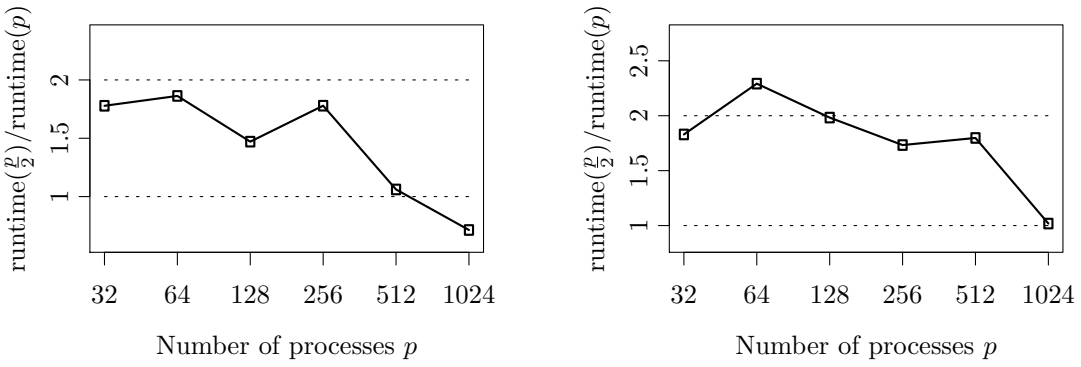


Figure 5.3.22.: Scaling results of the parallel local max algorithm on complete graphs with random edge weights. Left: complete graph with 499 991 253 edges. Right: complete graph with 3 999 980 403 edges. (performed on hc3)

Figure 5.3.23 shows the strong scaling results of the parallel local max algorithm for random geometric graphs of the 10th DIMACS Implementation Challenge. There is only a small speedup for the step when all cores of the nodes were used (8 to 16 processes). Especially for larger numbers of processes the scaling is *really good*, except for 256 processes in the left picture.

For the Delaunay graphs Figure 5.3.24 the speedup increases from no speedup to a speedup of about 1.8 (64 processes) constantly and then starts to fall. The parallel local

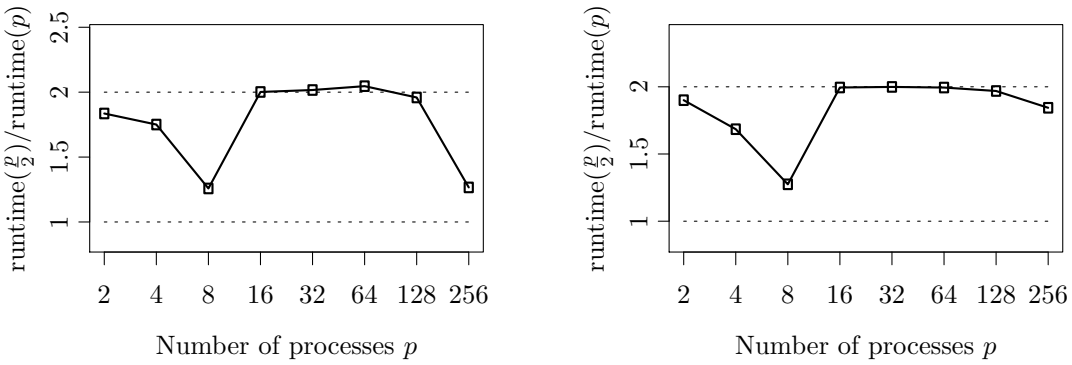


Figure 5.3.23.: Scaling results of the parallel local max algorithm on random geometric graphs with random edge weights. Left: rgg_n_2_23 (63 501 393 edges). Right: rgg_n_2_24 (132 557 200 edges).

max algorithm scales a lot *worse* for Delaunay graphs than for random geometric graphs, but still we see a speedup. This difference is probably due to the fact that our vertex block based partitioning method produces more *imbalanced* partitions for the Delaunay graphs than for the random geometric graphs and as we have seen in the weak scaling section (Section 5.3.1) the number of cross edges is a lot larger for Delaunay graphs than for random geometric graphs. If we look at the partitions for 16 processes of the rgg-n24 and delaunay-n25 graphs, we see that the smallest number of edges of a partition of the random geometric graph is 8 283 660 and the largest is 8 323 166. They just differ by a factor of 1.005. On the other hand the smallest partition of the Delaunay graph has 7 864 580 edges while the largest partition has 12 190 622 edges. That is a difference by a factor of 1.55.

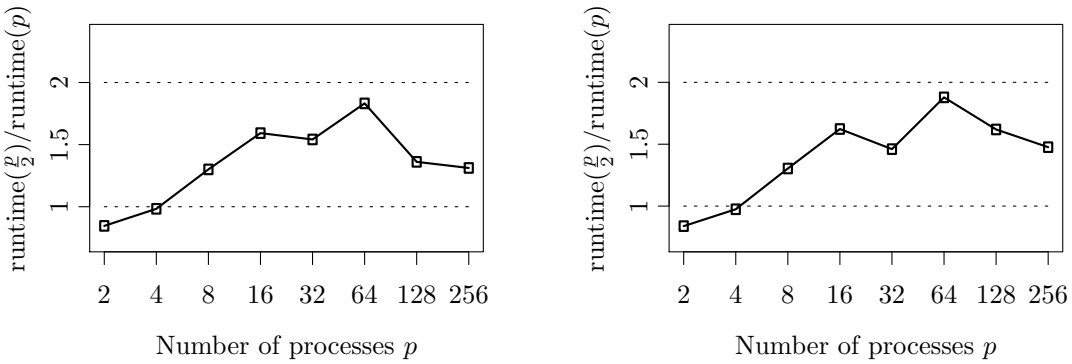


Figure 5.3.24.: Scaling results of the parallel local max algorithm on Delaunay graphs with random edge weights. Left: delaunay_n24 (50 331 601 edges). Right: delaunay_n25 (100 663 248 edges).

Figure 5.3.25 shows results for street graphs. The bad scaling for 8 processes is probably again because starting with 8 processes all of the available cores were used. The Europe-graph shows *good scaling* for 16 to 64 processes and the USA-graph for 64 to 256 processes. From what we have seen so far one would expect that the USA-graph is the larger one (scales better for more processes), but in fact the Europe-graph is almost twice as big. However the partitions of the USA graph are a lot more *imbalanced* than the partitions of the Europe-graph. In the case of 16 processes the USA graph has an imbalance factor of 1.52 (the quotient between the largest and the smallest partition) while the Europe-graph only has an imbalance factor of 1.18.

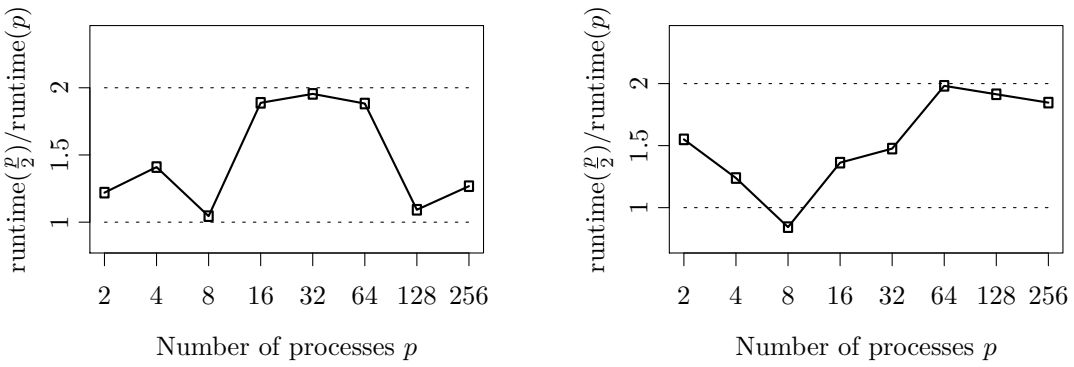


Figure 5.3.25.: Scaling results of the parallel local max algorithm on street graphs with random edge weights. Left: europe.osm (54 054 660 edges). Right: road_usa (28 854 312 edges).

For the web-graphs from Figure 5.3.26 the parallel local max algorithm *does not show good scaling*. It starts with quite a good scaling but then the speedups decrease, but the speedup is always larger than 1. Compared to the results of the parallel local tree algorithm that is really good.

Parallel Local Tree Algorithm

The strong scaling results for the parallel local tree algorithm are in general *similar* to the results that we have seen for the parallel local max algorithm. In case of the random geometric graphs and Delaunay graphs (Figure 5.3.32 and Figure 5.3.33) all available cores of the nodes were used starting with 32 processes, this probably explains the *drop in the speedup* at 32 processes.

The case were we see a *significant difference* compared to the results from the parallel local max algorithm is for web graphs (Figure 5.3.35). For the uk-2002 graph there is still a speedup larger than 1 but not as good as the speed up achieved by the parallel local max algorithm for the same graph. The results for the uk-2007 graph are even worse. For 128 to 1024 the runtime actually increases. The algorithm is for 1024 processes 1.47 times slower than for 16 processes.

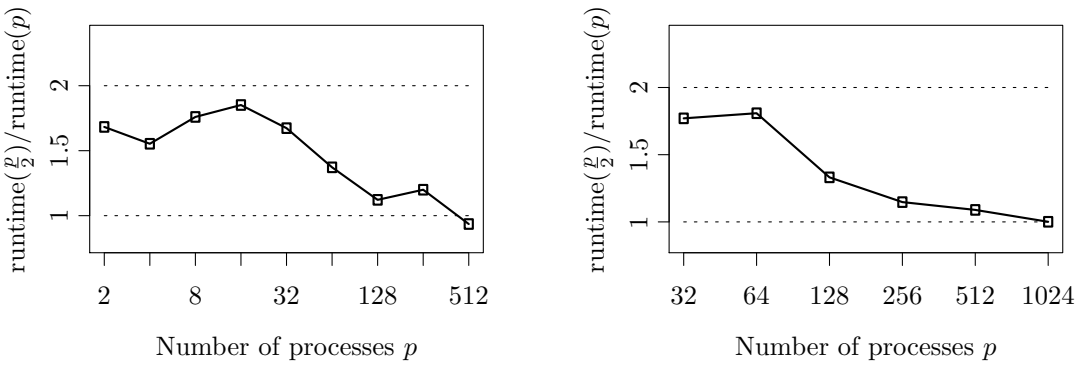


Figure 5.3.26.: Scaling results of the parallel local max algorithm on web graphs uk-2002 and uk-2007 (261 787 258 edge and 3 301 876 564 edges) with random edge weights. (right experiment performed on hc3)

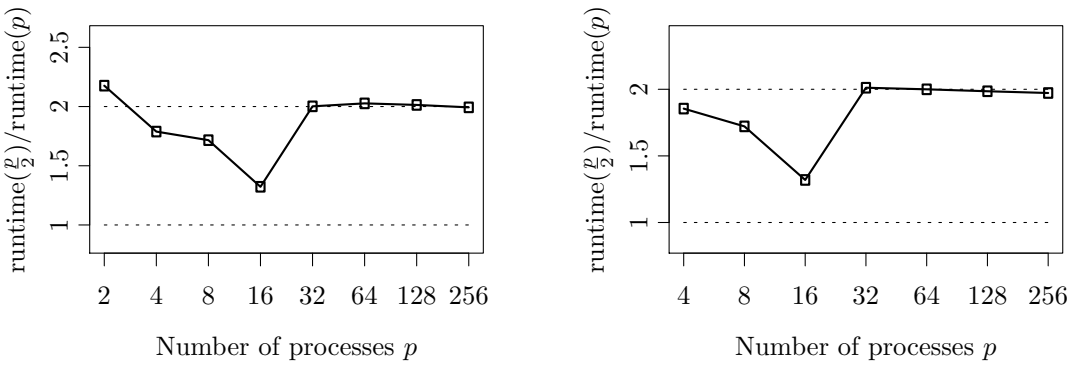


Figure 5.3.27.: Scaling results of the parallel local tree algorithm on 2D-grids with random edge weights. Left: 2D-grid with 49 990 000 edges. Right: 2D-grid with 199 980 000 edges.

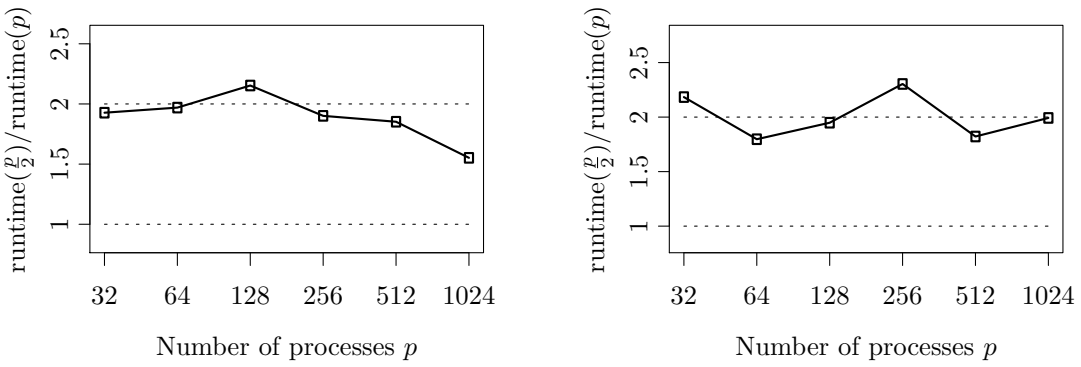


Figure 5.3.28.: Scaling results of the parallel local tree algorithm on 2D-grids with random edge weights. Left: 2D-grid with 500 007 064 edges. Right: 2D-grid with 4 000 025 124 edges. (performed on hc3)

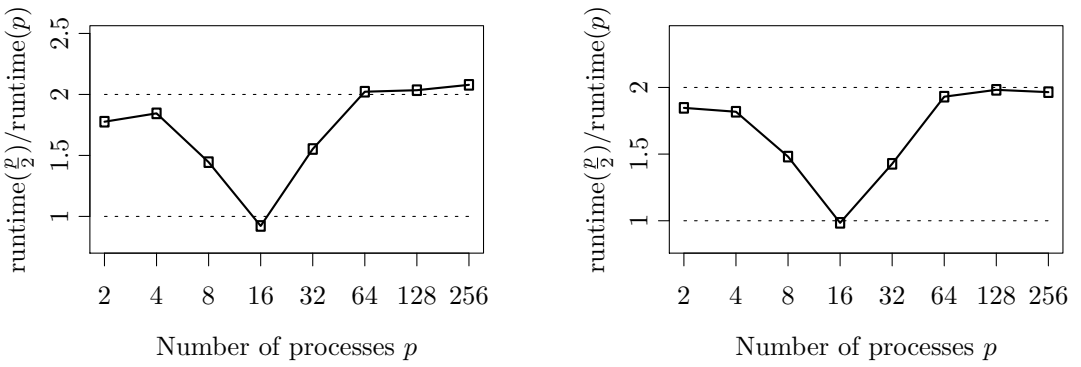


Figure 5.3.29.: Scaling results of the parallel local tree algorithm on 5D-grids with random edge weights. Left: 5D-grid with 46 875 000 edges. Right: 5D-grid with 189 747 360 edges.

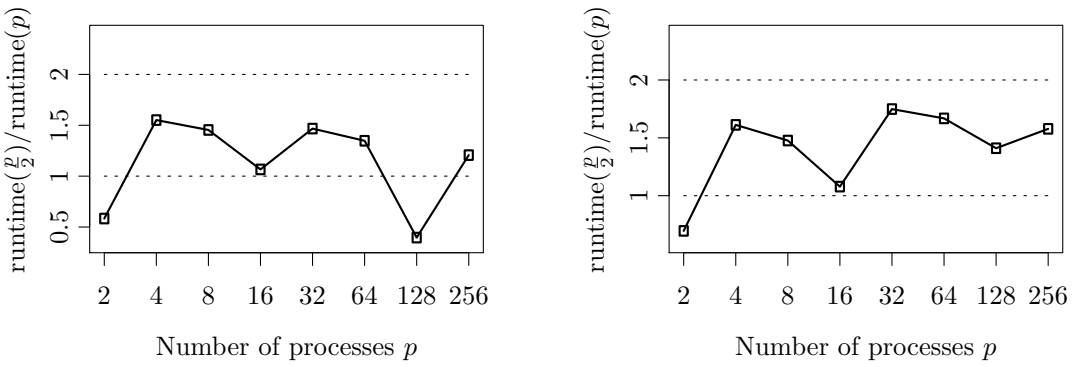


Figure 5.3.30.: Scaling results of the parallel local tree algorithm on complete graphs with random edge weights. Left: complete graph with 12 497 500 edges. Right: complete graph with 199 990 000 edges.

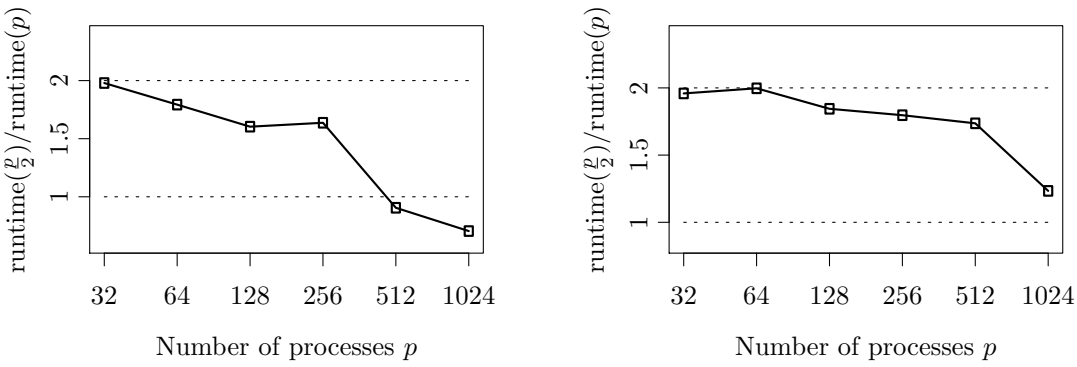


Figure 5.3.31.: Scaling results of the parallel local tree algorithm on complete graphs with random edge weights. Left: complete graph with 499 991 253 edges. Right: complete graph with 1 999 996 635 edges. (performed on hc3)

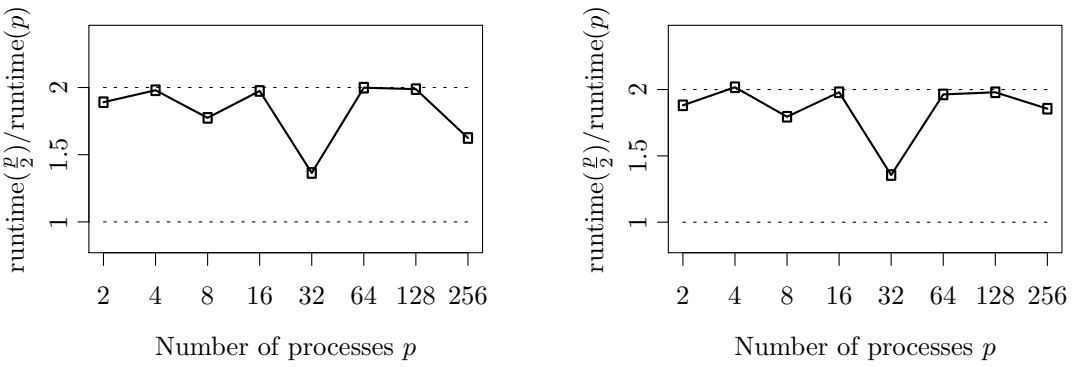


Figure 5.3.32.: Scaling results of the parallel local tree algorithm on random geometric graphs with random edge weights. Left: rgg_n_2_23 (63 501 393 edges). Right: rgg_n_2_24 (132 557 200 edges).

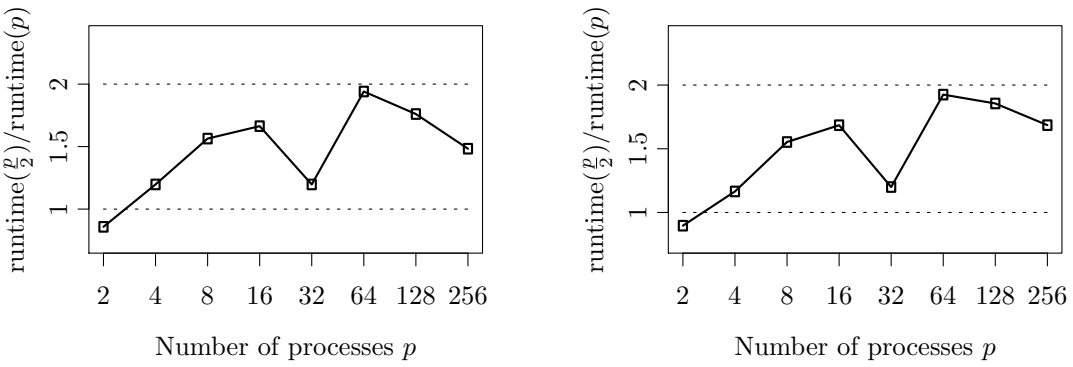


Figure 5.3.33.: Scaling results of the parallel local tree algorithm on Delaunay graphs with random edge weights. Left: delaunay_n24 (50 331 601 edges). Right: delaunay_n25 (100 663 248 edges).

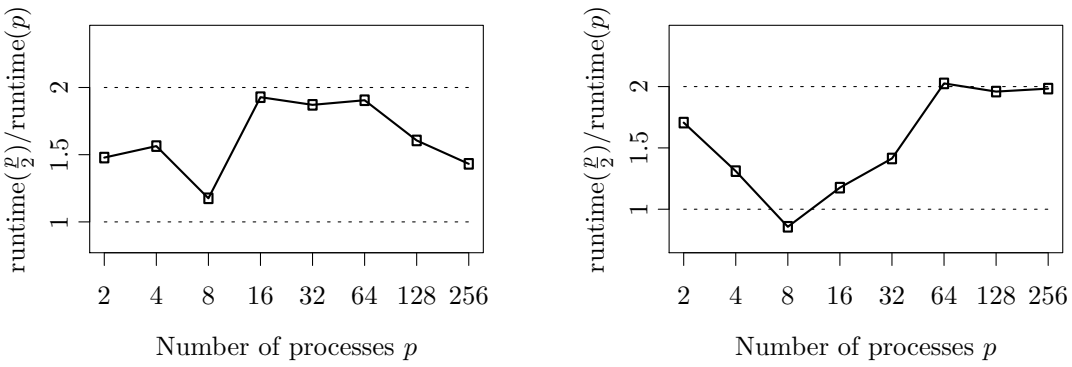


Figure 5.3.34.: Scaling results of the parallel local tree algorithm on street graphs with random edge weights. Left: europe.osm (54 054 660 edges). Right: road_usa (28 854 312 edges).

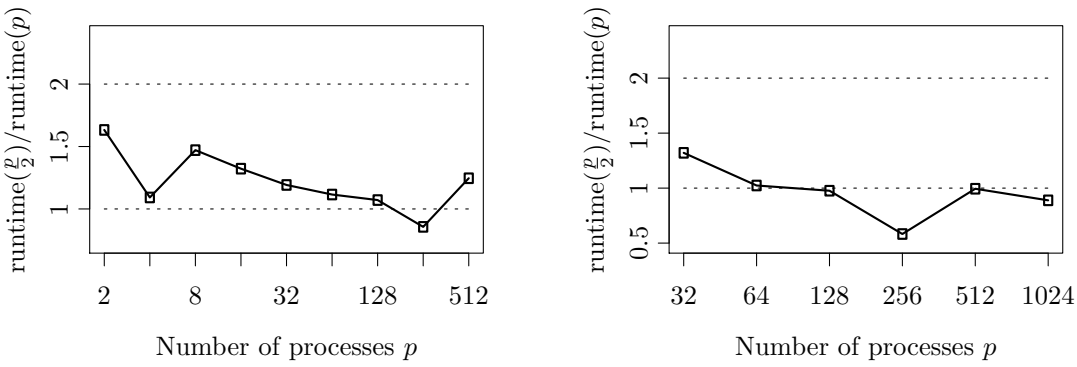


Figure 5.3.35.: Scaling results of the parallel local max algorithm on web graphs uk-2002 and uk-2007 (261 787 258 edge and 3 301 876 564 edges) with random edge weights. (right experiment performed on hc3)

Total Speedups

Tables 5.1 and 5.2 show the *total speedups* achieved by the parallel local max and parallel local tree algorithms for the experiments performed on the IC1. The speedups are computed using the runtimes of the sequential algorithms from Chapter 4. The second column of the tables shows the ratio between the sequential algorithms and the parallel algorithms using one process. As one can see the sequential algorithms are usually about 10%–20% faster than the parallel algorithms using one process.

The parallel local max algorithm achieves in all cases but one a speedup for 256 processes. The case where it does not achieve a speedup is for a complete graph with about 12 million edges, but this graph is a lot smaller than the other graphs. The *largest speedups* are achieved for 2D-grids and random geometric graphs. In case of 2d-grids we used an optimal partitioning, this results in far less cross edges. The number of cross edges for 2D-grids is only about 0.15%–0.3% of the total number of edges in case of 256 processes. In case of random geometric graphs we have that 2% and 1.5% of the edges are cross edges for 256 processes. On the other hand there are about 29% and 25% cross edges for the 5d-grids, about 40% cross edges for the Delaunay graphs and more than 99% of the edges of the complete graphs are cross edges. Those numbers are all for the partitions for 256 processes. This suggests that a better partitioning should result in better speedups. However in case of the europe.osm graph and uk_2002 graph we see speedups that are worse than the speedups of the 5D-grids, but the numbers of cross edges of those two graphs are 11% and 5%, respectively. Therefore the percentage of cross edges cannot be the only factor influencing the speedups.

Graph	Speedup 1 Process	Speedup 256 Processes
2D-grid, ∼50mio edges	0.89	87.0
2D-grid, ∼200mio edges	0.89	83.3
5D-grid, ∼47mio edges	0.82	28.1
5D-grid, ∼190mio edges	0.81	32.4
K_n, ∼12mio edges	0.97	0.9
K_n, ∼200mio edges	0.96	6.1
rgg_n_2_23	0.83	68.7
rgg_n_2_24	0.83	98.2
delaunay_n24	0.87	7.6
delaunay_n25	0.88	10.0
europe.osm	0.86	14.9
road_usa	0.87	19.9
uk_2002	0.92	24.3

Table 5.1.: Speedups for parallel local max algorithm.

The speedups achieved by the parallel local tree algorithm using 256 processes are in general better than the speedups achieved by the parallel local max algorithm, except for the uk_2002 graph. In case of the complete graphs the sequential algorithm required more time than the parallel version using only one process. This suggests that the

implementation of the sequential local tree algorithm is probably not optimal. In the case of complete graphs we used the runtime of the parallel algorithm to compute the speedup for 256 processes.

Graph	Speedup 1 Process	Speedup 256 Processes
2D-grid, ∼50mio edges	0.87	125.4
2D-grid, ∼100mio edges	0.87	122.4
5D-grid, ∼47mio edges	0.84	48.7
5D-grid, ∼190mio edges	0.86	45.2
K_n, ∼12mio edges	1.16	1.3
K_n, ∼200mio edges	1.03	11.5
rgg_n_2_23	0.85	98.9
rgg_n_2_24	0.84	111.2
delaunay_n24	0.86	14.0
delaunay_n25	0.84	16.5
europe.osm	0.85	36.5
road_usa	0.86	21.6
uk_2002	0.90	3.8

Table 5.2.: Speedups for parallel local tree algorithm.

5.3.3. Comparison Local Max and Local Tree

In this section we compare our two parallel algorithms directly, how they perform on different kinds of graphs with different amounts of processes. On the same kind of graphs they perform really similar, therefore we only chose one or two representative graphs for each kind.

Each figure shows the runtime of the two algorithms for different amounts of processes. We use a logarithmic scale for the y-axe to make it easier to distinguish the two curves. Additionally the figures also show the division of the runtime of the parallel local tree algorithm by the runtime of the parallel local max algorithm (dotted line).

The parallel local max algorithm performs in almost all cases (Figure 5.3.36 – Figure 5.3.44) *better* than the parallel local tree algorithm, except for complete graphs (Figure 5.3.39 and Figure 5.3.40), where the parallel local tree algorithm performs better. Also in most cases both algorithms scale pretty well to some extent, but they *do not show an optimal performance* (we have seen this in the previous section). There is one exception to this observation, in the case of the web graph uk-2007 (Figure 5.3.44) the parallel local tree algorithm performs really bad, there is actually no acceleration at all and the runtime gets larger for more processes.

There are two more observations regarding the factor f that the parallel local tree algorithm is slower than the parallel local max algorithm (except for the web graphs). The first observation is that this factor *tends to get smaller with larger numbers of processes*. And the other observations is that it also *gets smaller with a larger average vertex degree* (see Table 5.3). An explanation of this behaviour is that in a graph with a

larger average vertex degree we have proportionally less vertices than edges. The input size for the forest matching algorithm is less than the number of vertices and therefore the time spent on the forest matching algorithm might tend to be proportionally smaller than the time spent on computing candidates and removing edges for graphs with larger average vertex degrees. Those two operations depend on the number of edges and are the parts that both parallel algorithms have in common.

Graph	Average vertex degree	Range of factor f	Figure
europe.osm	~ 2.1	$6.12 - 3.44$	Figure 5.3.43
2D Graph	~ 4	$6.09 - 3.86$	Figure 5.3.36
Delaunay	~ 6	$4.53 - 2.96$	Figure 5.3.42
5D Graph	~ 9.7	$4.27 - 2.98$	Figure 5.3.38
Random geometric graph	~ 15.8	$2.81 - 2.14$	Figure 5.3.41
Complete Graph	19999	$0.55 - 1.1$	Figure 5.3.39

Table 5.3.: Average vertex degrees and runtime difference of the parallel local tree and parallel local max algorithms.

Those two observations suggest that the parallel local tree algorithm might be more suitable for graphs with *larger average vertex degrees* and for *larger numbers of processes*. Also as we previously mentioned the implementation of this algorithm is not optimal, there is an easier way to select a root of a parallel tree.

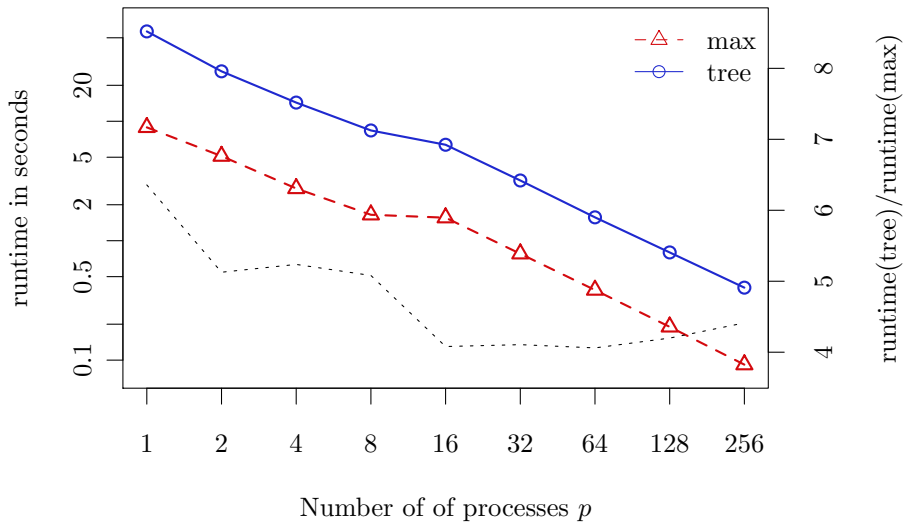


Figure 5.3.36.: Comparison of parallel local max and parallel local tree on 2D-grid with 99 983 940 edges.

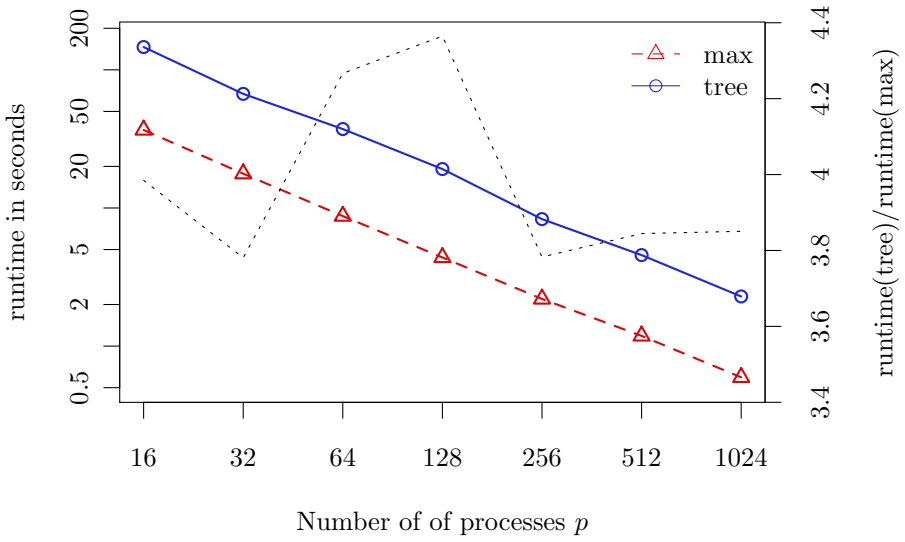


Figure 5.3.37.: Comparison of parallel local max and parallel local tree on 2D-grid with 4 000 025 124 edges. (performed on hc3)

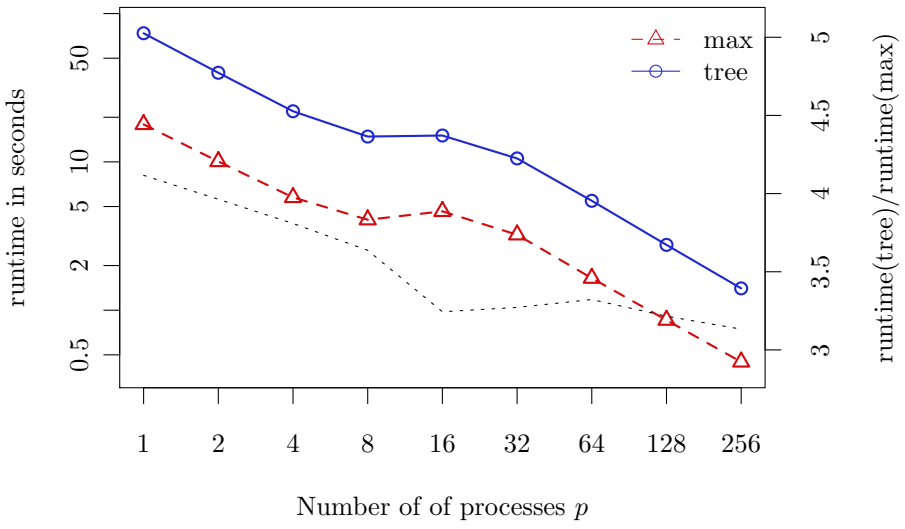


Figure 5.3.38.: Comparison of parallel local max and parallel local tree on 5D-grid with 189 747 360 edges.

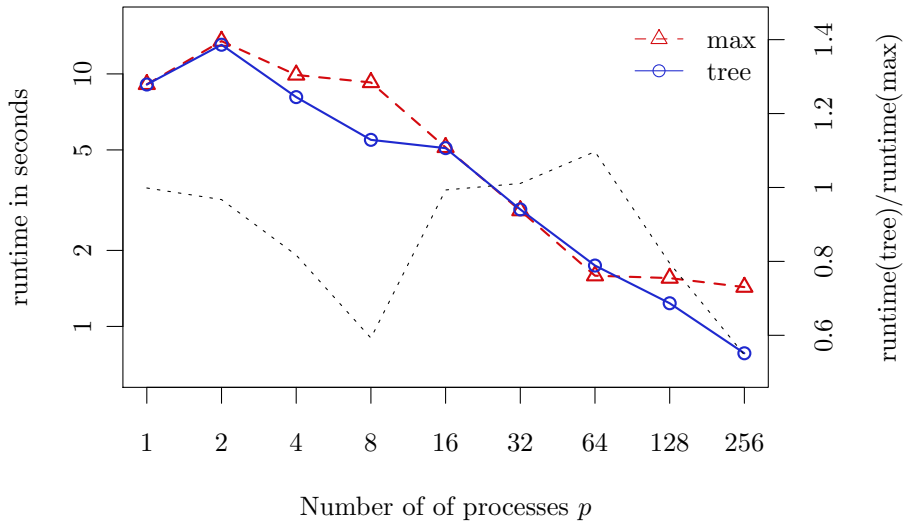


Figure 5.3.39.: Comparison of parallel local max and parallel local tree on complete graph with 199 990 000 edges.

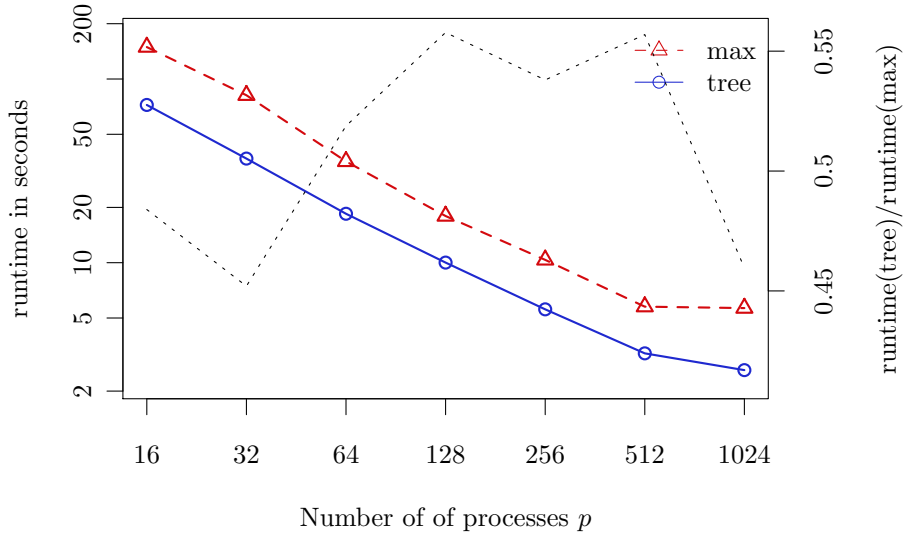


Figure 5.3.40.: Comparison of parallel local max and parallel local tree on complete graph with 3 999 980 403 edges. (performed on hc3)

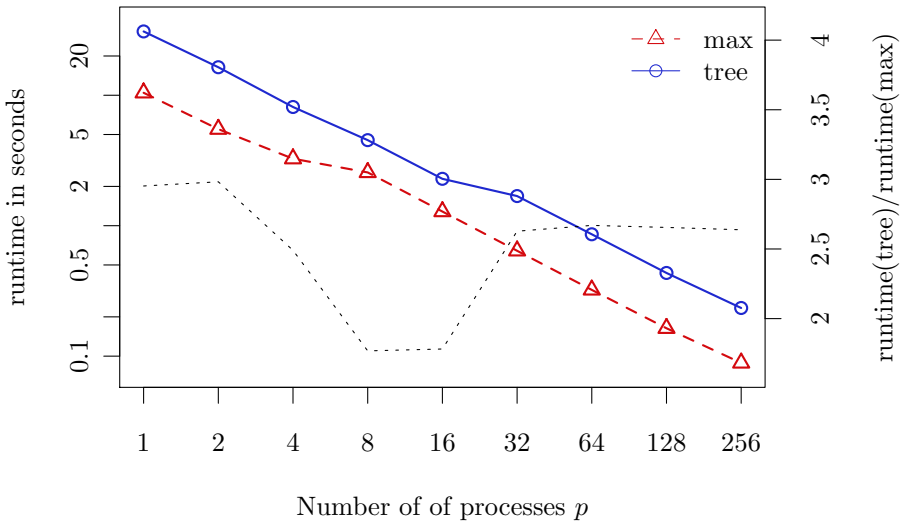


Figure 5.3.41.: Comparison of parallel local max and parallel local tree on random geometric graph (rgg_n_2_24, 132 557 200 edges).

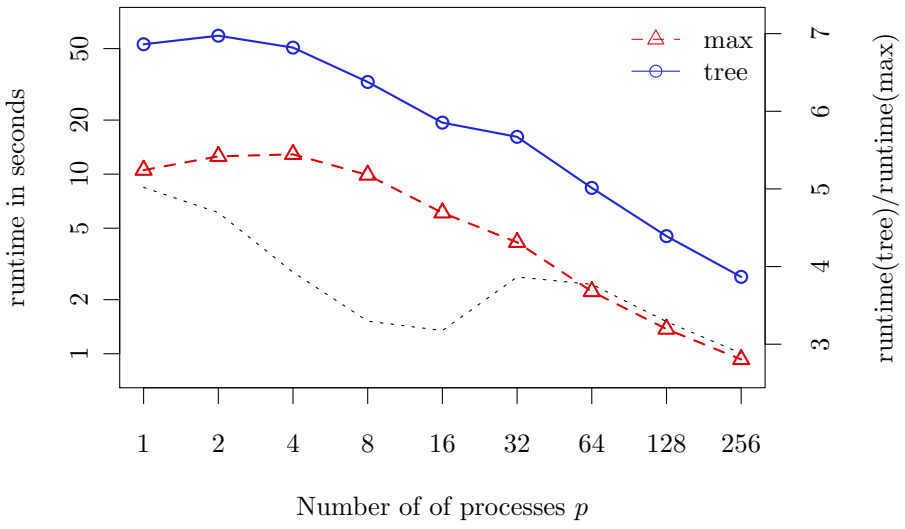


Figure 5.3.42.: Comparison of parallel local max and parallel local tree on delaunay graph (delaunay_25, 100 663 248 edges).

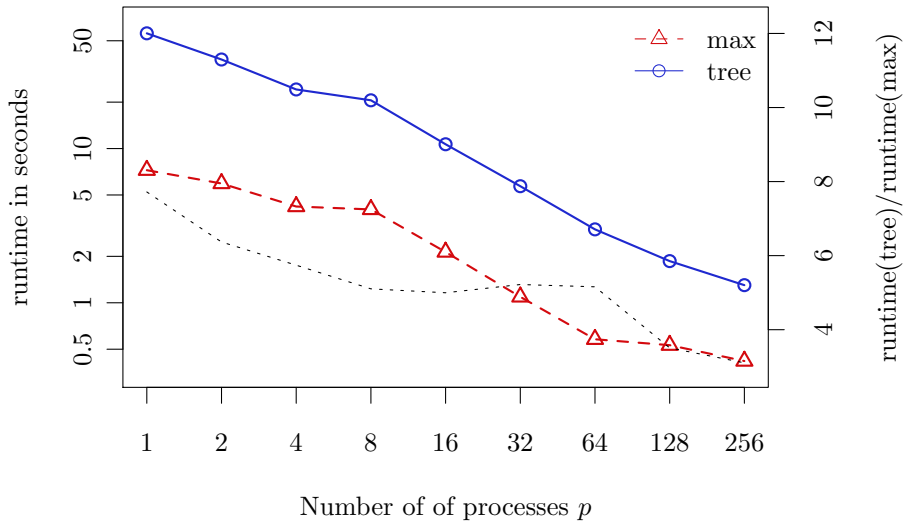


Figure 5.3.43.: Comparison of parallel local max and parallel local tree on street graph (europe.osm, 54 054 660 edges).

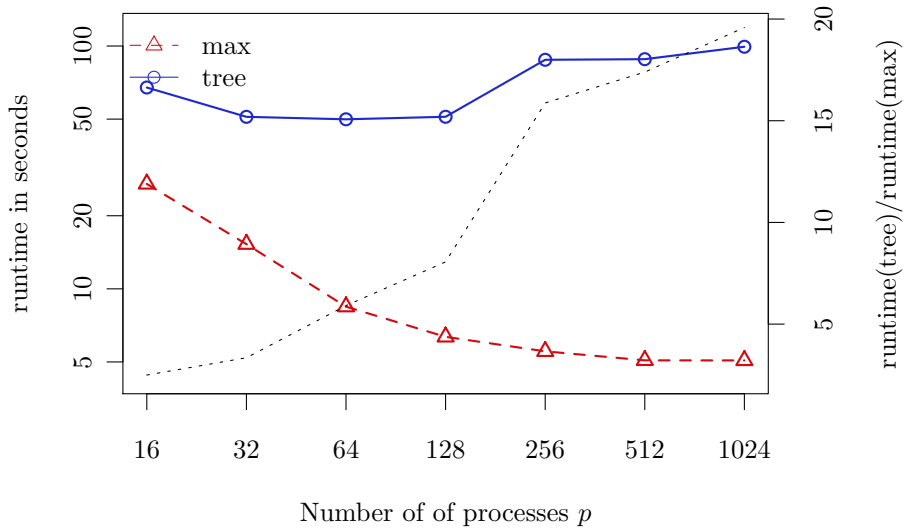


Figure 5.3.44.: Comparison of parallel local max and parallel local tree on web graph (uk-2007, 3 301 876 564 edges). (performed on hc3)

6. Conclusions

We presented three sequential matching algorithms. One that is based on the Karp-Sipser matching algorithm [21] but uses a different heuristic to select matched edges if there are no degree one vertices (the *mixed algorithm*). Another matching algorithm to compute approximate weighted matchings that is based on Preis' idea to match locally heaviest edges [30] (*local max algorithm*). The third algorithm computes for each vertex its heaviest incident edge and then computes maximum weighted matchings of the trees defined by these edges (*local tree algorithm*). All three algorithms have in common that they are round based which provides an intuitive way to parallelize them. We also introduced parallel version of the local max algorithm and the local tree algorithm.

We compared the three algorithms in experiments with GPA [26] and our own implementation of the Karp-Sipser algorithm with regard to the solution quality and the runtime. All three algorithms were on average within 8.5% of the solution of the Karp-Sipser algorithm for the cardinality problem. Karp-Sipser was on average 8.5% better than the local max algorithm, 4.7% than the local tree algorithm and about 2.5% better than the mixed algorithm.

In case of the GPA algorithm, which has been shown to compute matchings of good quality for the weighted matching problem [26], the differences in the solution quality are much smaller. GPA is in average about 1% better than the local max algorithm using the expansionstar2 rating. And for the same rating there is in average almost no difference between the local tree algorithm and GPA. The local tree algorithm, unlike GPA and the local max algorithm, does not provided any worst case guarantees for the quality of the solution. GPA and the local max algorithm are both $1/2$-approximation algorithms. Despite providing results with almost the quality of GPA or even the same quality, both algorithm were a lot faster than GPA. The local tree algorithm in average by a factor of 4 and the local max algorithm even by a factor of 15.

We also confirmed empirically the assumption that the local max algorithm and local tree algorithm require at most a logarithmic number of rounds. And we have shown that the runtime for slight variations of those two algorithms is expected to be linear for random edge weights.

For the parallel versions of the local tree algorithm and local max algorithm we performed extensive experiments to see how well they scale in practice. For those experiments we did not use any advanced graph partitioning technique, we just partitioned the graphs by assigning blocks of vertices to the processors. This resulted in optimal partitions for 2D-grids (by defining appropriate vertex IDs) and complete graphs. In case of nicely structured graphs (grids, random geometric graphs) we achieve good scaling qualities (strong and weak) for up to 1024 processors. The parallel algorithms performed reasonably well for complete graphs, which have a huge amount of cross edges, and for

Delaunay graphs, whose partition sizes differ quite a lot. For web graphs only the local max algorithm showed that speedups were achieved, the parallel local tree algorithm performed really bad on them.

6.1. Future Work

There are still several open questions regarding the real expected runtime of the algorithms. We have only shown an expected linear runtime for a variation of the algorithms where we assign new random edge weights at the start of each round to the remaining edges. It is still open if that is also the case without this modification of the algorithm. Our experimental results strongly suggest this assumption. In the experiments usually far more than half of the remaining edges were removed during a round.

An important property for the parallel local tree algorithm is that none of the local trees get too large and more importantly not too deep. So far we have checked this property empirically, but it still remains to show that this property holds in theory (for random edge weights).

We only used a simple approach to partition the input graphs for the parallel algorithms. So there is potential that the parallel algorithms perform better when using advanced graph partitioners, especially in the case of the Delaunay and web graphs.

For the parallel local tree algorithm it still remains to implement the easier method to compute a root of a parallel tree. This adjustment should result in shorter runtimes but we do not expect a different behaviour, e.g. better scaling results for the web graphs. Because the parallel dynamic programming part to compute matchings of parallel trees is not that different from the current approach to compute the root of a parallel tree.

A. Zusammenfassung

Die Berechnung von Matchings von Graphen tritt in der Informatik häufig als Teilproblem eines größeren Problems auf. Zum Beispiel bei der Berechnung von Graphpartitionen.

Ein *Matching* M eines Graphen $G = (V, E)$ ist eine Teilmenge der Kantenmenge E des Graphen, so dass keine zwei Kanten aus M einen gemeinsamen Endknoten haben. Bei *Kardinalitätsmatchings* versucht man die Anzahl an Kanten in M zu maximieren und im Fall von *gewichteten Matchings* versucht man das Gesamtgewicht zu maximieren.

Bereits in der 1960er Jahren hat Edmonds gezeigt, dass das Berechnen von optimalen Kardinalitätsmatchings bzw. optimalen gewichteten Matchings in polynomieller Zeit möglich ist [13]. Die besten Algorithmen zur Bestimmung von optimalen Matchings haben eine Laufzeit von $O(n(m + n \log n))$ [17]. In der Praxis sind diese Algorithmen häufig aber trotzdem zu langsam und optimale Ergebnisse nicht unbedingt notwendig. Deshalb gibt es inzwischen ein starkes Interesse an approximativen Algorithmen mit deutlich kürzeren Laufzeiten. Zum Beispiel der Karp-Sipser Algorithmus zur Lösung des Kardinalitätsproblems mit einer linearen Laufzeit [21] oder der LAM Algorithmus von Preis [30] und GPA Algorithmus von Maue und Sanders [26] für das gewichtete Problem. Beide Algorithmen erreichen eine $1/2$-Approximation. LAM hat eine lineare Laufzeit und GPA eine Laufzeit von $O(m \log n)$.

Diese Arbeit behandelt Variationen von bereits existierenden sequentiellen Matching-Algorithmen sowie einen neuen sequentiellen Matching Algorithmus. Der *Mixed Algorithmus* ist eine Variante des Algorithmus von Karp und Sipser, benutzt aber eine andere Heuristik bei nicht optimalen Entscheidungen. Der *Local Max Algorithmus* basiert auf der Idee vom LAM Algorithmus lokal schwerste Kanten auszuwählen. Der *Local Tree Algorithmus* ist ein neuer Matching Algorithmus für das gewichtete Problem bei dem zuerst die schwersten inzidenten Kanten von Knoten ausgewählt werden und dann optimale gewichtete Matchings berechnet werden für die Bäume die durch diese Kanten definiert werden. Für den Local Max Algorithmus und den Local Tree Algorithmus werden auch parallele Versionen vorgestellt.

Die sequentiellen Algorithmen werden sowohl theoretisch als auch experimentell untersucht. So wird gezeigt, dass die erwartete Laufzeit eines leicht veränderten Local Max Algorithmus sowie Local Tree Algorithmus, für zufällige Kantengewichte, linear ist. Die Veränderung besteht darin am Anfang einer jeden Runde der Algorithmen neue zufällige Kantengewichte für die verbleibenden Kanten zu berechnen.

In Experimenten werden die drei Algorithmen mit dem Algorithmus von Karp und Sipser sowie dem GPA Algorithmus verglichen. Die Experimente zeigen, dass alle drei Algorithmen, für eine große Auswahl an unterschiedlichen Graphen, durchschnittlich innerhalb von 8.5% vom Ergebnis vom Karp-Sipser Algorithmus sind für Kardi-

naliätsmatchings. Dabei sind die Ergebnisse vom Mixed Algorithmus besser als vom Local Tree Algorithmus und der Local Tree Algorithmus produziert bessere Ergebnisse als der Local Max Algorithmus.

Im Fall von gewichteten Matchings werden Ergebnisse für das expansionstar2 Rating aus [20] präsentiert. In diesem Fall werden der Local Max und der Local Tree Algorithmus mit dem GPA Algorithmus verglichen für den bereits gezeigt wurde, dass er in der Praxis gewichtete Matchings mit einer guten Qualität berechnet [26]. Hier sind Ergebnisse deutlich enger beieinander. Der Local Max Algorithmus ist durchschnittlich nur 1% schlechter als GPA und im Vergleich von GPA und dem Local Tree Algorithmus sieht man durchschnittlich keine Qualitätsunterschiede.

Für die sequentiellen Algorithmen Local Max und Local Tree wird außerdem noch die lineare Laufzeit experimentell überprüft und wie groß die erwartete Anzahl an Runden ist.

Bei den parallelen Algorithmen wird experimentell deren Skalierungsverhalten überprüft. Dabei zeigt sich, dass in den meisten Fällen ab einer größeren Anzahl von Prozessoren eine gute "schwache Skalierung" beobachtet werden kann. Im Fall "starker Skalierung" sind die Ergebnisse für schön strukturierte Graphen gut (Gitter-Graphen, zufällige geometrische Graphen). Aber auch für andere Graphtypen zeigt sich eine Beschleunigung der Laufzeit wenn mehr Prozessoren verwendet werden. Für die parallelen Experimente wurden bis zu 1024 Prozessoren verwendet und die Graphgrößen variieren von 20 Millionen Kanten bis zu 4 Milliarden Kanten.

B. LAM and Unique Edge Weights

There is an interesting property of the LAM-algorithm when using unique edge weights.

Theorem B.1. *For a graph G with unique edge weights Algorithm 3.1.2 returns the same set of matched edges every time it is run on G. That is the resulting set is independent from the order in which we add locally heaviest edges to the matching.*

Before proving Theorem B.1 we introduce alternating sequences and an observation about those sequences and matchings.

Definition B.1. *An* alternating sequence $s = \langle s_1, \ldots, s_i \rangle$ *of a matching M is a sequence of edges such that each $s_j \in M$ $(j = 1, \ldots, i)$ and two consecutive edges s_j and s_{j+1} are connected by another edge of the graph G.*

A decreasing alternating sequence $s = \langle s_1, \ldots, s_i \rangle$ *is an alternating sequence such that $\omega(s_j) > \omega(s_{j+1})$ and for the connecting edge e of two consecutive edges s_j and s_{j+1} we have $\omega(s_j) > \omega(e) > \omega(s_{j+1})$.*

Figure B.0.1 shows an example for a decreasing alternating sequence. The matching produced by Algorithm 3.1.2 is $M = \{s_1, s_2, s_3\}$ and the decreasing alternating sequence of edge s_3 is $s = \langle s_1, s_2, s_3 \rangle$.

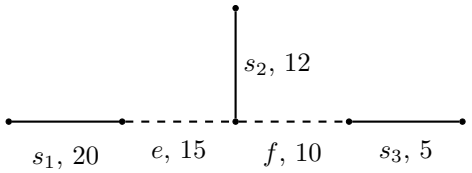


Figure B.0.1.: Decreasing alternating sequence example.

Lemma B.1. *For each edge e of a matching M of a graph G obtained by Algorithm 3.1.2 there is a* decreasing alternating sequence $s = \langle s_1, \ldots, s_i \rangle$*, such that s_1 is a locally heaviest edge in G and $s_i = e$.*

Proof. Let the edge e be an edge of the matching M. If e is a locally heaviest edge in G then we have found such a sequence $s = \langle e \rangle$.

If e is not a locally heaviest edge in G then there must be an unmatched adjacent edge f of e with $\omega(f) > \omega(e)$. Because we are running Algorithm 3.1.2 and f is unmatched there must be another edge e' that is adjacent to f (and not to e) which is matched and which is heavier than f $(\omega(e') > \omega(f))$.

Now there are two possibilities either e' is a locally heaviest edge in G, then we have found the sequence, if it is not we repeat the same procedure to find another edge which is heavier and connected to e' by an unmatched edge which is heavier than e'.

This construction must end at a certain point since there is only a finite amount of edges in G and the newly constructed edges of the sequence are heavier than the old ones.

□

Now we are able to proof Theorem B.1:

Proof. Let M and M' be two different matchings produced by two runs of Algorithm 3.1.2 on graph G. Obviously one matching cannot be a subset of the other matching, otherwise one of them would not be maximal.

Since both matchings are different and not a subset of the other one, there must be an edge $e \in M$ and $e \notin M'$. According to Lemma B.1 there is a decreasing alternating sequence $s = \langle s_1, \ldots, s_i = e \rangle$ in M for edge e. Obviously s is not contained in M' because $e = s_i \notin M'$. Let s_j $(1 < j \leq i)$ be the first edge of sequence s such that $s_j \notin M'$. The edge s_1 must be in M' since it is a locally heaviest edge in G.

Because $s_j \notin M'$ there must be an edge $e' \in M'$ adjacent to s_j with $\omega(e) > \omega(s_j)$. Again using Lemma B.1 there is a decreasing alternating sequence $s' = \langle s'_1, \ldots, s'_k = e' \rangle$ for edge e' in M'. The sequence s' cannot be contained in M otherwise e would not be an edge of M. Thus there is an edge s'_m $(1 < m \leq k)$ of s' with $\omega(s'_m) > \omega(s_j)$ and $s'_m \notin M$. This let us construct another decreasing alternating sequence in M which is not in M' analog to the construction from before. Repeating this construction of sequences which are in one of the matchings but not in the other finally leads to a decreasing alternating sequence of size one. That is because the newly created sequences end on edges with higher weight then the last edges of sequences created before them and we only have a finite amount of edges!

But since this final sequence consists of a single edge, this edge must be a locally heaviest edge in G and thus must be contained in both matchings M and M'. This contradicts the assumption that the sequence is not contained in both matchings! Hence the first assumption that $M \neq M'$ must be wrong.

□

C. Parallel Local Max Details

C.1. How to Break Ties

In the sequential local max algorithm (Algorithm 4.1.2) we used edge IDs to break ties, because in this case computing edge IDs is really simple, they are just given by the order in which we add edges to the graphs. For the parallel version of the local max algorithm (Algorithm 5.1.1) using edge IDs to break ties is more complicated, because we add edges to graphs on different processes. Hence the edge IDs would not be unique anymore and in the case of cross edges there might be different IDs assigned to a single edge, because it is located on more than one process. Also the format used to store graphs by the 10th DIMACS Implementation Challenge [5] does not define edge IDs in any natural way (it is an adjacency list like structure). Therefore we decided to use the global vertex IDs to break ties in the parallel case. In case of a tie between the edges $e = \{u, v\}$ and $f = \{w, x\}$ we say the edge e is heavier iff:

$$\max(u, v) > \max(w, x) \vee (\max(u, v) = \max(w, x) \wedge \min(u, v) > \min(w, x))$$

In the case of multi edges it would not be clear which edge to choose, thus we decided not to allow multi graphs, to keep everything simple. Also we are not able to use the local IDs of vertices to break ties, this would result in similar problems as we have seen with edge IDs which are defined by the order in which they are added to the subgraphs. Figure C.1.1 shows an example of one of the problems that might occur when using local IDs instead of global IDs.

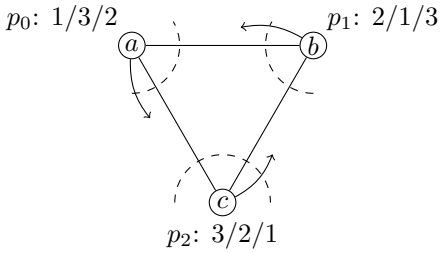


Figure C.1.1.: Problem when using local vertex IDs to choose heaviest incident edge. First entry of the triple is the vertex ID of process p_0, second of p_1 and third of p_2.

Each of the three vertices is located on a different process (p_0, p_1 or p_2) and the triples next to each vertex specify the local IDs from the perspective of the different processes. Assuming that each edge has the same weight, then the heaviest incident edge of vertex

a is $\{a, c\}$ (from process p_0's perspective), of vertex b it is the edge $\{b, a\}$ and of vertex c it is the edge $\{c, b\}$. In this example no edge would be chosen as a locally heaviest edge and therefore the algorithm would not terminate.

C.2. How to Receive Messages

We tested two variations how to receive messages within Algorithm 5.1.1 using MPI. MPI allows us to receive messages in several ways, e.g. there are blocking and non blocking receive operations. There are also options (e.g. MPI_Probe) to check if there is an incoming message of a specific type or from a specific process.

The first variant was to probe (MPI_Probe) for messages of a specific type from an active partner. This option not only returns if there is such a message but it also returns the size of the incoming message. Knowing the size of a message allows us to allocate the correct amount of memory necessary to receive the message. After the allocation of the memory we use a blocking receive operation to receive the messages.

The other option that we tested was to use a non blocking receive operation (MPI_Irecv), for all possible incoming messages, i.e. one receive operation for each active partner. The problem with this implementation is that we need to provide a receive buffer that is large enough to store the incoming message form a partner. As described in Section 5.1.1 a process does not know if the messages from partners actually contain any data, and even if an incoming message contains data, the process cannot know the size of this message. But each process knows the maximum possible size of an incoming message of a partner p, that is the number of ghost vertices to this partner. Therefore we decided to set the size of the receive buffers to this number. Each non blocking receive operation returns a status object which allows us to check if a receive operation has finished. There are actually operations to check if one particular receive is done, or only some of a set of receive operations or even if all receive operations have finished. We decided to use the "some" version, so we are able to set the candidates of ghost vertices as soon as possible but also to minimize the test for finished receive operations.

Using non blocking receive operations we hoped to get a better parallelism while receiving messages. But our experiments showed that runtime differences where minimal (see Table C.1). Therefore we decided to use the probe version, which uses less memory.

Graph	Processes	Irecv	Probe
Delaunay_n25	8	6.83608	6.37075
Delaunay_n25	16	4.47492	4.3211
Rgg_n24	8	1.73904	1.68922
Rgg_n24	16	0.908357	0.856891
UK 2002	8	2.81231	2.71366
UK 2002	16	1.69961	1.65273
k_{8000}	8	0.849289	0.842611
k_{8000}	16	0.469272	0.452277
2D-grid, $n = 8000^2$	8	1.32723	1.33043
2D-grid, $n = 8000^2$	16	0.655326	0.663498

Table C.1.: Comparison of receiving messages using non blocking receive and probe-version. Runtime in seconds.

D. Detailed Parallel Local Tree

D.1. Detailed Computation of a Parallel Forest

We describe in this section how to compute parallel trees from a set of edges in more detail. In particular we describe the general approach how to decide on a global root vertex for a parallel tree and how to create the trees of a forest based on those roots.

The decision process on *computing roots* of parallel trees is based on border components as outlined in Section 5.2.3. So the first step is to compute the border components. Border components are represented by the following structure. A border component consists of a list of *partners*. A partner represents a cross edge to a border component on another process. Each partner object stores the process ID of the other process and the IDs of the local vertex and ghost vertex of the represented cross edge. A border component also stores the ID of a root candidate and the IDs of the local vertex and ghost vertex of the cross edge that points in the direction of the root candidate, we call this edge the *root edge* of the border component. Algorithm D.1.1 outlines the procedure how to compute the border components, it also returns the roots of purely local trees. Border components are defined by connected components of the set of local trees. We use a union-find structure, to compute the connected components. Initially each local vertex is the representative of its own component. Two components are combined if they are connected by an edge. This is done by the first for-loop (line 1). Each of the computed connected components is represented by one of its vertices (*uf.find*(...)).

Some of those connected components might be incident to a cross edge which connects this component with a component on another process. Those are the border components. The second for-loop (line 4) computes the border components. For each cross edge $e = \{v, u\}$ we get the representative of the connected component of the local vertex v (as before the first mentioned vertex of a cross edges is the local vertex and the second one the ghost vertex). We set this representative to be visited, we use this information in another step to easily identify border components. But more important we use the representative to get the border component object *bc* of this connected component. Afterwards we add a partner, representing the cross edge e, to the border component *bc* and also add *bc* to the list of all border components (*bcs*) of the current process, if we have not done this before. After iterating all cross edges we have added to each border component all necessary partners.

We only marked the representatives of border components as visited, all the other connected components do not have any incident cross edges. Therefore all unvisited representatives represent purely local trees and we can use them as the roots of those trees (last for-loop of Algorithm D.1.1).

We have already outlined the basic idea for the decision process of the global root of

Algorithm D.1.1 Compute border components of a parallel forest

get_border_components(*local_edges*, *cross_edges*, *local_roots*, *bcs*):

```
 1: for all e = {v, u} ∈ local_edges do
 2:    uf.union(v, u)
 3:
 4: for all e = {v, u} ∈ cross_edges do
 5:    rep = uf.find(v)
 6:    set rep to visited
 7:    bc = border_component(rep)
 8:    bc.add(Partner(proc_of(u), v, u))
 9:    if bc ∉ bcs then
10:       bcs.add(bc)
11:
12: for all e = {v, u} ∈ local_edges do
13:    rep = uf.find(v)
14:    if not visited rep then
15:       local_roots.add(rep)
16:       set rep to visited
```

a parallel tree in Section 5.2.3. We omitted to specify which vertex is chosen by each border component to be its candidate, we decided to use the vertex with the *smallest* ID of all the end vertices of cross edges of the border component. We also omitted how to determine when the decision process terminates.

Because no border component is able to know the size of the parallel tree, they cannot just stop after the candidate has not changed for a certain amount of rounds. Consider the border component B_2 of Figure D.1.1, after B_2 sent its candidate ID to B_1 it is not possible that B_2 informs B_1 in another round about a new candidate that B_1 does not know. All candidate changes of B_2, in following rounds, happen because B_1 informed B_2 about this candidate. We say that B_2 is an *outer component* of B_1. More generally speaking a border component B_a is an *outer component* of a border component B_b if B_a receives no messages from another border component.

We said that after B_2 sent its candidate to B_1 it will not send any new candidates to B_1, but this message is actually not necessary because B_2 only has one cross edge and B_1 knows this cross edge, therefore it knows the candidate of B_2. This is the case for all initial outer components, those are the leaves of the border component tree. Therefore all initial outer components can send the message $\langle$*no more messages*$\rangle$ to their partners and do not have to inform them about their candidate.

Also a component B_a which is initially not an outer component of another component B_b, can send the message $\langle$*no more messages*$\rangle$ to B_b, as soon as it knows that it is an outer component of B_b. The component B_a informed B_b in an earlier round about any candidate that it did not receive from B_b.

As soon as a border component receives the message $\langle$*no more messages*$\rangle$ from all of its partners it knows the ID of the root vertex and also from which direction it received

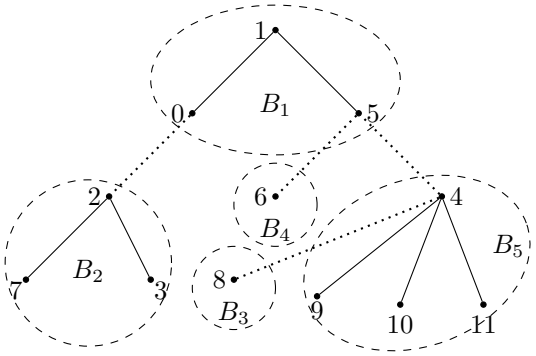


Figure D.1.1.: Border components of Figure 5.2.4

the message about this vertex for the first time. This border component is done with the computation of the root after sending ⟨*no more messages*⟩ to all the remaining partners.

The whole process on deciding on a root vertex for each parallel tree of a forest is shown in Algorithm D.1.2. We assume that the initial candidate of each border component has already been computed. The first for-loop computes the messages that each border component sends to the neighboring border components. Again we add each message to a send buffer of each target process, to reduce the number of MPI-messages. The second for loop then sends those bundled messages to the targets. The last for-loop is used to receive messages from partners and updates the states of border components depending on the received messages. The algorithm returns as soon as there are no more components which still need to send or receive messages.

As one can see each partner-object now has two more states. The first one (*is active target*) is used to decided whether we still need to send messages to this partner component and the second one (*is active source*) is used to decide if we still receive messages from this partner component. The first state also tells us if we have already sent the message ⟨*no more messages*⟩ to the partner and both states also help us to decide if the algorithm is allowed to terminate. The algorithm is allowed to terminate if the process no longer receives any messages or sends any messages to another component.

There is one check to see if a border component *bc* still receives messages from a partner other than a partner p (line 3). This is the check if *bc* is an outer component of the border component of p. One way to implement this check is by adding a counter to the partner-object. This counter is initially set to the number of other partners of the corresponding component. Whenever we receive the message ⟨*no more messages*⟩ at one partner we reduce the counter of all other partners by 1. Then if the counter is 0, we know we do not receive any new messages from other partners.

The active target and source processes can be computed during the first for-loop, by checking the states of the individual partners of the border components.

After the termination of Algorithm D.1.2 each border component knows the global root and the root edge specifies the direction from where a border component was informed about global root for the first time, therefore it is the edge with the shortest distance to

Algorithm D.1.2 Compute the root vertices of parallel trees of a forest

decide_on_root(*bcs*):

```
 1: for all border components bc ∈ bcs do
 2:   for all partners p ∈ bc do
 3:     if bc does not receive messages from partners other than p then
 4:       if  p is active target then
 5:         set p to inactive target
 6:         msgs_for_proc[proc_of(p)].add(⟨no more messages, p⟩)
 7:     else
 8:       msgs_for_proc[proc_of(p)].add(⟨candidate_of(bc), p⟩)
 9:
10: for all active target processes proc do
11:   send msgs_for_proc[proc] to process proc
12:
13: for all active source processes proc do
14:   incoming_msgs = receive messages from proc
15:   for all m ∈ incoming_msgs do
16:     if m of type ⟨no more messages, p⟩ then
17:       set p to inactive source
18:     else if m of tpye ⟨c, p⟩ then
19:       bc = border_component_of(p)
20:       if c < candidate_of(bc) then
21:         candidate_of(bc) = c
22:         root_edge_of(bc) = p
23:
24: if active targets and sources still exist then
25:   decide_on_root(bcs)
```

the root and hence the root of the local tree. Each root edge $e = \{u, v\}$ is obviously a cross edge, but only for one border component the local vertex u of e is the root of the parallel tree. For this border component the local vertex u is the root of the local tree and for all other components the ghost vertex v of the root edge is the root of the local tree.

Like in the sequential case we added all the provided local and cross edges to a graph, the forest, and the trees are built using a breadth first search starting at the root vertices. In the case of local trees that is the same as in the sequential case, we just iterate over all local roots (computed by Algorithm D.1.1) and call the function *build_tree* from the sequential section.

Building the subtrees of parallel trees is done based on the selection of the roots that we have just described, Algorithm D.1.3 shows this procedure. To build the subtrees we can use a slightly different version of the function *build_tree* from Section 4.3. We allow to specify the parent of a vertex (the last argument to the function). Otherwise

we might walk in the wrong direction.

Algorithm D.1.3 Build the parallel trees of a forest

build_parallel_trees(*forest*, *bcs*):

```
1:  roots = ∅
2:  for all border components bc ∈ bcs do
3:     e = {u, v} = root_edge_of(bc)
4:     if u = candidate_of(bc) then
5:        roots = roots ∪ u
6:        build_tree(forest, u, u)
7:     else
8:        roots = roots ∪ v
9:        build_tree(forest, u, v)
10: return roots
```

We need to be able to specify the parent of a vertex, otherwise it would be possible to create damaged trees. Consider the example from Figure D.1.2. The first picture shows the whole parallel tree, the second one only the tree from process 0s perspective and the last picture the tree from process 1s perspective. On process 0 there are actually two border components, one for the vertex 0 and one for vertex 1. Those two components are connected by the cross edges $\{1, 2\}$ and $\{0, 2\}$. The global root of this tree is the vertex 0, but if we are not able to specify a parent vertex the function call *build_tree*(*forest*, 1) would create a tree on process 0 whose root is vertex 1 and not vertex 0. Which might be really problematic for larger trees. So instead we make a function call *build_tree*(*forest*, 1, 2) which tells the function that the parent of vertex 1 is vertex 2. An edge from vertex 1 to 2 would be deleted from the adjacency list of vertex 1 and not followed.

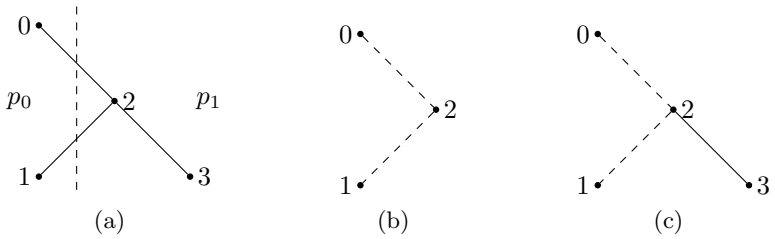


Figure D.1.2.: Subtrees connected by cross edges.

D.1.1. Example: Computation of Border Components

The computation of border components is based on connected components defined by the local edges. Figure D.1.3 shows the computation of the connected components using a union-find structure for the example tree from Figure D.1.1.

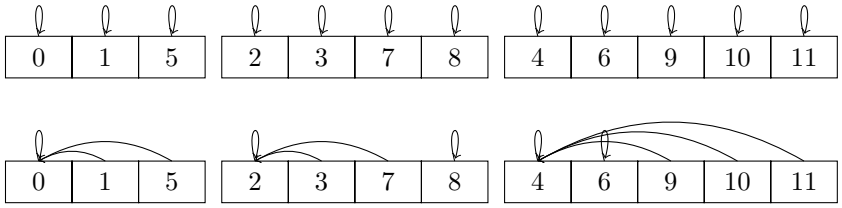


Figure D.1.3.: Computing connected components of the local edges from Figure 5.2.4 using union-find.

Initially each vertex points to itself, indicating that each vertex is in its own connected component. After combining all connected components, which are connected by a local edge, we get the result shown in the bottom picture of Figure D.1.3. There are five vertices left which point to themselves. Those are the representatives of the five border components which we have seen in Figure 5.2.5. Now all that is missing is to add the partners to each border component and to compute candidates of the border components. The triples (partners) $(1, 0, 2)$, $(2, 5, 4)$ and $(2, 5, 6)$ are added to the border component B_1 represented by 0. The first entry of each triple is the process ID of the partner process and the second and third entries are the local vertex and the ghost vertex of the represented cross edge. For the remaining border components we get $B_2 = \{(0, 2, 0)\}$, $B_3 = \{(2, 8, 4)\}$, $B_4 = \{(0, 6, 5)\}$ and $B_5 = \{(0, 4, 5), (1, 4, 8)\}$. The candidates of the border components are: Vertex 0 for B_1 and B_2, vertex 4 for B_3 and B_5 and vertex 5 for B_3.

D.1.2. Example: Decide on Global Root

The example from the previous sections results in the following situation:

- The candidate of B_1 is 0 at the edge $(0, 2)$
- The candidate of B_2 is 0 at the edge $(2, 0)$
- The candidate of B_3 is 4 at the edge $(8, 4)$
- The candidate of B_4 is 5 at the edge $(6, 5)$
- The candidate of B_5 is 4 at the edge $(4, 5)$

During the first round of deciding on a root process the following messages are sent:

- B_1 sends the message $\langle 0 \rangle$ to B_2, B_4 and B_5
- B_2 sends $\langle \textit{no more messages} \rangle$ to B_1
- B_3 sends $\langle \textit{no more messages} \rangle$ to B_5
- B_4 sends $\langle \textit{no more messages} \rangle$ to B_1

- B_5 sends $\langle 4 \rangle$ to B_1 and B_3

And we have the following receive operations:

- B_1 receives $\langle$*no more message*$\rangle$ from B_2 and B_4, therefore it becomes an outer component of B_5. It also receives $\langle 4 \rangle$ from B_5 which does not cause a candidate update.
- B_2 receives $\langle 0 \rangle$ from $B_1 \Rightarrow$ no candidate update
- B_3 receives $\langle 4 \rangle$ from $B_5 \Rightarrow$ no candidate update
- B_4 receives $\langle 0 \rangle$ from $B_1 \Rightarrow$ no candidate update
- B_5 receives $\langle 0 \rangle$ from $B_1 \Rightarrow$ new candidate is 0 at edge $(4, 5)$, also receives $\langle$*no more messages*$\rangle$ from B_3, therefore it becomes an outer component of B_1

Now we are done with the first round and we get the following situation for the second round:

- B_1 sends $\langle 0 \rangle$ to B_2 and B_4 and $\langle$*no more messages*$\rangle$ to B_5
- B_5 sends $\langle 0 \rangle$ to B_3 and $\langle$*no more messages*$\rangle$ to B_1

This results in the following receive operations:

- B_1 receives $\langle$*no more messages*$\rangle$ from $B_5 \Rightarrow$ becomes an outer component of B_2 and B_4
- B_2 receives $\langle 0 \rangle$ from $B_1 \Rightarrow$ no candidate update
- B_3 receives $\langle 0 \rangle$ from $B_5 \Rightarrow$ new candidate is 0 at edge $(8, 4)$
- B_4 receives $\langle 0 \rangle$ from $B_1 \Rightarrow$ no candidate update
- B_5 receives $\langle$*no more messages*$\rangle$ from $B_1 \Rightarrow$ becomes an outer component of B_3

During the last round the components send the following messages:

- B_1 sends $\langle$*no more messages*$\rangle$ to B_2 and B_4
- B_5 sends $\langle$*no more messages*$\rangle$ to B_3

And the following messages are received:

- B_2 receives $\langle$*no more messages*$\rangle$ from B_1
- B_3 receives $\langle$*no more messages*$\rangle$ from B_4
- B_4 receives $\langle$*no more messages*$\rangle$ from B_1

Now each border component knows that the vertex 0 is the root and it also knows which partner (cross edge) sent the information about the root vertex for the first time. This cross edge is the root of the local tree of this border component.

D.2. Detailed Parallel Dynamic Programming

In this section we describe the implementation of the dynamic programming part of the computation of maximum weighted matching of parallel forest in more detail.

Algorithm D.2.1 shows the implementation of the *bottom up* part of computing maximum weighted matchings of parallel trees. At first we compute for each subtree (each root) the *number of dependencies*, that is the number of subtrees this subtree depends on (the number of leaves which are ghost vertices). We do not provide the implementation of this function, but it can be done using a breadth first search through the subtree. After this computation we have an initial iteration over all roots and check if the corresponding subtree has no dependencies. In such a case we compute the result of this subtree and add it to the list of messages of the parent. The computation of the results is done by using the function *fill_subtree_table_of_root* which corresponds to the fill subtree table function from the sequential algorithm, but it stops as soon as it reaches a ghost vertex and also returns a bundle of messages. This functions fills for each vertex the array *outgoing_edge*, which states for a vertex which of the outgoing edges is matched if the incoming edge is not matched. The returned bundle of messages contains one message for each child c of the root r of the subtree. The messages have the form $\langle result,\ c\rangle$. The *result* is a tuple containing the two possible weights of the subtree starting at the child c. The first weight is for the cases that the edge $\{c, r\}$ is matched and the second weight is for the case the edge is not matched. Afterwards we send all new messages to the corresponding processes and then start to receive incoming messages. Unlike for the parallel local max algorithm we will not send empty messages. Because we do not expect messages from particular sources, we just receive any incoming messages and we know the total number of incoming messages (that is the number of cross edges from child subtrees), therefore we know when we can stop waiting for messages.

Algorithm D.2.1 Compute the results of the subtrees

```
fill_subtree_table_parallel(forest, roots_of_parallel_trees):
 1: set_number_of_dependencies(forest, roots_of_parallel_trees)
 2: for all roots r ∈ roots_of_parallel_trees do
 3:    if number_of_dependencies[r] == 0 then
 4:       msg = fill_subtree_table_of_root(r)
 5:       msgs_of_proc(proc_of(r)).add(msg)
 6:
 7: send_messages(msgs_of_proc)
 8:
 9: while not all roots handled do
10:    msgs_of_proc = receive_msgs_and_fill_subtree_table(forest)
11:    send_messages(msgs_of_proc)
```

The function *receive_msgs_and_fill_subtree_table* (Algorithm D.2.2) waits for at least one incoming message, that is the minimum number of messages required before the process can do any new computations. As soon as it receives a bundle of messages

of the form $\langle result,\ v\rangle$ it sets the result of the ghost vertex v, reduces the number of dependencies of the corresponding subtree, identified by the root r, by one and if the subtree no longer has any dependencies the result for this subtree is computed. In this case we add a message for the parent subtree to the list of messages for the process of the parent. Obviously we only add the message if r is not the root of the parallel tree, otherwise there would not be a parent. We receive incoming messages as long as there are any, to compute a maximal number of new message and thus reducing the total number of MPI-messages.

Algorithm D.2.2 Receive incoming messages and compute results of subtrees

receive_msgs_and_fill_subtree_table(*forest*):

```
 1: repeat
 2:    msgs = receive_incoming_message()
 3:
 4:    for all messages ⟨result, v⟩ ∈ msgs do
 5:       r = root_of[v]
 6:       number_of_dependencies[r] -= 1
 7:       subtree_result[v] = result
 8:       if number_of_dependencies[r]==0 then
 9:          msg = fill_subtree_table_of_root(r)
10:          if r is not the root of the parallel tree then
11:             msgs_of_proc(proc_of(r)).add(msg)
12: until no incoming message
13:
14: return  msgs_of_proc
```

Filling the array *outgoing_edge* stops as soon as there are no more subtrees which have any dependencies. One simple way to implement this check, is by using a simple counter.

Algorithm D.2.3 shows the implementation *top down* phase, this phase adds matched edges to the matching. Initially we compute for each root subtree the matched edges and messages to their child subtrees. Additionally we also compute for each process the total number of messages that this process will receive (number of cross edges to parents).

The function *add_matched_edges_and_msgs_for_children* (Algorithm D.2.4) adds the edges of a maximum weighted matching of the tree starting at the given root r to the resulting matching M. The additional information that this function requires is if the incoming edge of r is matched or not, we have seen this in the sequential tree matching algorithm. But there is one essential difference to the sequential algorithm, as soon as the algorithm reaches a ghost vertex v it stops and computes a message of the form $\langle v,\ matched\_incoming\rangle$ and adds this message to the message bundle of the process $p(v)$ of ghost vertex v. All this message does is to tell process $p(v)$ if the incoming edge of the vertex v is matched, that is all that is required to compute the matched edges of the subtree starting at v on $p(v)$. The array *outgoing_edge* states which of the outgoing edges of a vertex is matched if the incoming edge is not. For more information about this see Section 3.3.

Algorithm D.2.3 Get the matched edges of the parallel forest

get_matched_edges_of_parallel_forest(*forest*, *roots_of_parallel_trees*):

```
1:  M = ∅
2:  n = 0 // number of incoming messages
3:  for all roots r ∈ roots_of_parallel_trees do
4:    if r is root of parallel tree then
5:      // r cannot be a ghost vertex
6:      add_matched_edges_and_msgs_for_children(forest, r, false, M, msgs_of_proc)
7:    else
8:      n += out_degree(r)
9:
10: send_messages(msgs_of_proc)
11:
12: while n > 0 do
13:   receive_msgs_and_add_matched_edges(forest, n, M, msgs_of_proc)
14:   send_messages(msgs_of_proc)
15:
16: return M
```

Algorithm D.2.4 Add the matched edges of a subtree to the matching and compute the necessary messages

add_matched_edges_and_msgs_for_children(*forest*, *r*, *matched_incoming*, *M*, *msgs_of_proc*):

```
1:  if r is ghost vertex then
2:    msgs_of_proc[proc_of(r)].add(⟨r, matched_incoming⟩)
3:  else
4:    if matched_incoming then
5:      for all outgoing_neighbors(r, forest) n do
6:        add_matched_edges_and_msgs_for_children(forest n, false, M)
7:    else
8:      for all outgoing_neighbors(root, T) n do
9:        if n == outgoing_edge[r] then
10:         M = M ∪ {r, n}
11:         add_matched_edges_and_msgs_for_children(T, n, true, M)
12:       else
13:         add_matched_edges_and_msgs_for_children(T, n, false, M)
```

After the initial computation of matched edges of the root subtrees Algorithm D.2.3 we send the computed messages to their targets and then start to receive messages until all required messages have been received. The receive procedure is shown in Algorithm D.2.5. Again we receive at least one message, that is the minimum number that is required before a process can continue with computations, using a blocking receive. For each received message $\langle v, i \rangle$ we reduce the number of remaining incoming messages

and start adding matching edges using the information i whether the incoming edge of v is matched.

Algorithm D.2.5 Receive messages from parents and compute matched edges

receive_msgs_and_add_matched_edges(*forest*, n, M, *msgs_of_proc*):

```
1: repeat
2:    msgs = receive_incoming_message()
3:    n -= size_of msgs // reduce number of incoming messages
4:    for all messages ⟨v, i⟩ ∈ msgs do
5:       // v is not a ghost vertex!
6:       add_matched_edges_and_msgs_for_children(forest, v, i, M, msgs_of_proc)
7: until no incoming message
```

Like in Section 5.1 whenever we send information about a vertex we have to transform the local ID to the corresponding global ID and vice versa when receiving messages. Also whenever we have to wait for an incoming message we use this time to compute maximum weighted matchings of purely local trees. After we have computed the matchings of all parallel trees we compute the matchings of the remaining unprocessed local trees.

Bibliography

[1] Message Passing Interface. http://www.mcs.anl.gov/research/projects/mpi/.

[2] Valgrind. http://valgrind.org.

[3] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[4] David Avis. A survey of heuristics for the weighted matching problem. *Networks*, 13:475–493, 1983.

[5] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. 10th DIMACS Implementation Challange. http://www.cc.gatech.edu/dimacs10/.

[6] Brenda S. Baker. Approximation Algorithms for NP-Complete Problems on Planar Graphs. *Journal of the ACM*, 41(1):153–180, 1994.

[7] Umit Catalyurek, Florin Dobrian, Assefaw Gebremedhin, Mahantesh Halappanavar, Alex Pothen, and Pacific Northwest. Distributed-Memory Parallel Algorithms for Matching and Coloring. In *Workshop on Parallel Computing and Optimization PCO*, pages 1–12, 2011.

[8] Martin Dietzfelbinger, Hendrik Peilke, and Michael Rink. A More Reliable Greedy Heuristic for Maximum Matchings in Sparse Random Graphs. *arXiv:1203.4117v1*, pages 1–17, March 2012.

[9] Doratha E. Drake and Stefan Hougardy. A Simple Approximation Algorithm for the Weighted Matching Problem. *Information Processing Letters*, 85:211–213, 2003.

[10] Doratha E. Drake and Stefan Hougardy. Linear Time Local Improvements for Weighted Matchings in Graphs. In *In: Proceedings of the 2nd International Workshop on Experimental and Efficient Algorithms (WEA-03). Volume 2647 of LNCS.*, pages 107–119. Springer-Verlag, 2003.

[11] I.S. Duff and J Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications*, 22(4):973–996, 2001.

[12] David Padua (Ed.). Encyclopedia of Parallel Computing, 2011.

[13] Jack Edmonds. Maximum Matching and a Polyhedron With 0,1-Vertices. *Physics*, 69(June):125–130, 1965.

[14] Steinbuch Centre for Computing. InstitutsCluster (IC1). http://www.scc.kit.edu/dienste/4945.php, April 2012.

[15] Steinbuch Centre for Computing. KIT-Hochleistungsrechner HP XC3000. http://www.scc.kit.edu/dienste/hc3.php, 2012.

[16] M. Fujii, T. Kasami, and K. Ninomiya. Optimal sequencing of two equivalent processors. *SIAM Journal on Applied Mathematics*, 17(4):784–789, 1969.

[17] Harold N. Gabow. Data Structures for Weighted Matching and Nearest Common Ancestors with Linking. *Proceedings of the first annual ACM-SIAM Symposium on Discrete Algorithms*, pages 434–443, 1990.

[18] John L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, May 1988.

[19] Jaap-Henk Hoepman. Simple distributed weighted matchings. *arXiv:cs/0410047v1*, pages 1–7, 2004.

[20] Manuel Holtgrewe, Peter Sanders, and Christian Schulz. Engineering a scalable high quality graph partitioner. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2010.

[21] Richard M. Karp and M. Sipser. Maximum Matchings in Sparse Random Graphs. In *Proceedings of the 22nd IEEE Annual Symposium on Foundations of Computer Science*, pages 364–375. IEEE, October 1981.

[22] Eugene L. Lawler. *Combinatorial Optimization : Networks and Matroids*. Holt, Rinehart and Winston, 1976.

[23] Michael Luby. A simple parallel algorithm for the maximal independent set problem. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, volume 15 of *STOC '85*, pages 1–10. ACM, 1985.

[24] Jakob Magun. Greeding matching algorithms, an experimental study. *Journal of Experimental Algorithmics (JEA)*, 3, September 1998.

[25] Fredrik Manne and Rob Bisseling. A Parallel Approximation Algorithm for the Weighted Maximum Matching Problem. In *Parallel Processing and Applied Mathematics*, volume 4967 of *Lecture Notes in Computer Science*, pages 708–717. Springer Berlin / Heidelberg, 2008.

[26] Jens Maue and Peter Sanders. Engineering algorithms for approximate weighted matching. In *Proceedings of the 6th International Conference on Experimental Algorithms*, pages 242–255. Springer-Verlag, 2007.

[27] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures : The Basic Toolbox*. Springer, Berlin, 2008.

[28] Md. Mostofa Ali Patwary, Rob H. Bisseling, and Fredrik Manne. Parallel greedy graph matching using an edge partitioning approach. In *Proceedings of the 4th International Workshop on High-level Parallel Programming and Aplications*, HLPP '10, pages 45–54, New York, New York, USA, 2010. ACM Press.

[29] Seth Pettie and Peter Sanders. A simpler linear time $2/3-\varepsilon$ approximation for maximum weight matching. *Information Processing Letters*, 91(6):271–276, September 2004.

[30] Robert Preis. Linear Time $\frac{1}{2}$-Approximation Algorithm for Maximum Weighted Matching in General Graphs. In *Proceedings of the 16th Annual Conference on Theoretical Aspects of Computer Science*, STACS'99, pages 259–269. Springer-Verlag, 1999.

[31] Peter Sanders and Christian Schulz. Engineering Multilevel Graph Partitioning Algorithms. In Camil Demetrescu and Magnús Halldórsson, editors, *Algorithms – ESA 2011*, volume 6942 of *Lecture Notes in Computer Science*, pages 469–480. Springer Berlin / Heidelberg, 2011.

[32] Maximilian Schuler. *Engineering Edge Ratings and Matching Algorithms for Multilevel Graph Partitioning Algorithms*. Bachelor's thesis, Karlsruhe Institute of Technology, 2011.

[33] A. J. Soper, C. Walshaw, and M. Cross. A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph-Partitioning. *Journal of Global Optimization*, 29(2):225–241, June 2004.

[34] Robert L. Thorndike. The problem of classification of personnel. *Psychometrika*, 15(3):215–235, 1950.

[35] Thomas Wang. Integer Hash Function. http://www.concentric.net/~ttwang/tech/inthash.htm, March 2007.